Blinker

für

Clipper

```
  Title: Blinker 3.3 Online Reference
Credits:    __   __   (C) 1990-95 Assembler Software Manufacturers Inc.
           (..) (..)  Blink Inc, 8001 W. Broad, Richmond, VA 23294
                .     Sales    : (804) 747-6700 BBS : (804) 747-7333
             \___/    Technical: (804) 747-3600 FAX : (804) 747-4200
                      FaxFacts : (804) 747-5333
```

BLINKER OVERLAY PAGEFRAME   Enable use of the EMS pageframe for overlays

---

Purpose : **Enable use of the EMS pageframe for overlays**

Syntax   : **BLINKER OVERLAY PAGEFRAME ON | OFF**

Default : **OFF**

Example : **# Use the EMS pageframe**
BLINKER OVERLAY PAGEFRAME ON

Note: Please read this section fully before using this command.This command will allow the overlay manager to utilize the expanded memory pageframe for the overlay pool (maximum size 64kb), on machines equipped with LIM EMS (3.2 or higher) expanded memory boards or emulators.

Note: This command will be ignored at runtime in the absence of an EMS driver (or pageframe).

To use this option there must be 64kb or more of available expanded memory, and the program's overlay OPSIZE must be less than 64kb. At runtime, if expanded memory is available and this option is enabled, the overlay manager will allocate 4 expanded memory pages, map them into the pageframe, and use the pageframe for overlay execution. The overlay OPSIZE will automatically be expanded to use all 64kb available in the pageframe, resulting in increased execution speed of overlaid code.

WARNING: When using this command, your program MUST NOT use ANY functions which access expanded memory via the EMS pageframe UNLESS the pageframe mapping state is preserved, and you make sure no overlaid functions are called prior to restoring the mapping state.

Additionally, overlaid functions will not be able to access expanded memory via the EMS pageframe UNLESS they take particular care to determine within which page(s) they are executing, otherwise they may map themselves out of memory. Failure to adhere to these guidelines will result in program failure, corruption of the program's expanded memory, or both. Also be aware that some TSRs, network drivers and disk caches do not save/restore the pageframe mapping state correctly, which may lead to program failure when overlays are executed in the pageframe. The use of the pageframe may be adjusted at runtime through the use of the BLINKER environment variable (see the 'Environment Variable' section in chapter 2).

If both this command and the UMB command are used, and both the pageframe and a UMB are available, Blinker will use the pageframe in preference.

If either or both of these are specified and are not available on the end user's system, Blinker will automatically ignore these commands and allocate the overlay area from conventional memory.

Notes for CA-Clipper Summer '87:

This command is only effective when the fixed overlay allocation method is selected with the command BLINKER OVERLAY FIXED. When this command is activated with CA-Clipper Summer'87, Blinker will disable CA-Clipper's use of expanded memory by automatically burning in the E0 setting to the CLIPPER environment variable.

Notes for CA-Clipper 5.x:

When this command is activated with CA-Clipper 5.x, Blinker will automatically burn in the BADCACHE setting to the CLIPPER environment variable, causing CA-Clipper 5.x to save the pageframe mapping state between each access to expanded memory. This will protect the overlay area in the pageframe from corruption.

See Also:

BLINKER OVERLAY THRESHOLD   Set size of the smallest overlaid segments

---

Purpose : **Set size of the smallest overlaid segments**

Syntax   : **BLINKER OVERLAY THRESHOLD <nuLimit>**

Default : **32 bytes**

Example : **# Only overlay segments larger than 64 bytes**
BLINKER OVERLAY THRESHOLD 64

This command sets the threshold size below which code specified to be overlaid will be moved back to the root section of the program. Very small routines (less than 32 bytes) take up more root space when they are overlaid, due to overlay manager table entries, than they would if they were simply left in the root.

Additionally, very small overlaid routines execute disproportionately slower, since the time overhead in loading / calling them may be large in comparison with the actual execution time of the routine. If it is the case that very small routines are very frequently called, the developer may wish to specify a threshold large enough to cause them to be moved back to the root, improving the performance of the program.

CA-Clipper 5.x note

This command has no effect on CA-Clipper 5.x code overlaid through the paging system, but could be used, for example, for C functions within a CA-Clipper program.

BLINKER OVERLAY UMB **Enable use of the XMS UMBs for overlays**

Purpose : **Enable use of the XMS UMBs for overlays**

Syntax : **BLINKER OVERLAY UMB ON | OFF**

Default : **ON**

Example : **# Do not use XMS UMB's for the overlay area**
BLINKER OVERLAY UMB OFF

This command used in conjunction with a suitable XMS driver allows the Blinker overlay manager to execute overlays in upper memory blocks above 640kb. Blinker's use of UMBs is dependent on there being a suitably large contiguous block available. The size of the UMB allocated will be the same as the program's overlay opsize.

Use of this command requires a 386 processor (or better), and an XMS 2.0 extended memory driver that supports UMB's (upper memory blocks). UMB's are blocks of memory which are mapped into the area above 640kb, using the memory management capabilities of the 386/486 family of processors. Upper memory blocks are sometimes referred to as 'high memory' or LOADHI areas.

If your XMS driver supports the loading of drivers into high memory, it is likely that you can use this feature. Blinker will need a UMB at least as large as the overlay area, so you may need to alter the placement of your programs in high memory to facilitate this. Most 386 memory managers provide utilities to help optimize this area. If both this command and the PAGEFRAME command are used, and both the pageframe and a UMB are available, then Blinker will use the pageframe.

If either or both of these are specified and are not available on the end user's system, Blinker will automatically ignore these commands and allocate the overlay area from conventional memory. This use of UMB's may be adjusted at runtime through the use of the BLINKER environment variable (see the 'Environment Variable' section in Chapter 2).

Notes for CA-Clipper Summer '87:

This command is only effective when the fixed overlay allocation method is selected with the command BLINKER OVERLAY FIXED.

See Also:

BLINKER PROCEDURE DEPTH **Specify the maximum depth of CA-Clipper procedure nesting**

Purpose : **Specify the maximum depth of CA-Clipper procedure nesting**

Syntax : **BLINKER PROCEDURE DEPTH <nuDepth>**

Default : **50**

Example : **# Increase stack size for 3rd party libraries**
BLINKER PROCEDURE DEPTH 60

This command is provided as a convenience to CA-Clipper programmers to allow the setting of the program stack based on the maximum procedure nesting depth which will occur in the application at runtime. This depth is the number of procedure calls that are executed before a procedure return. For example, if procedure A calls procedure B which calls procedure C, the maximum depth is 3. But, if procedure B returns to procedure A which then calls procedure C, the maximum depth is 2.

The default depth is 50 procedures, but this may be increased if it is possible for more nesting to occur. For instance, a recursive procedure (one that calls itself) will often require a greater depth. The size of the stack set is approximately 1kb per 10 levels. Thus the default of 50 procedures has an overhead of approximately 5kb, and this will rise proportionally as the PROCEDURE DEPTH is increased.

Since both this command and the STACK command control the size of the application stack, only one of these commands should be used (the last one specified is the one that is processed). Programmers using languages other than CA-Clipper should use the STACK command directly, since the calculation that Blinker performs to compute the final stack size with this command is based on the stack requirements for CA-Clipper procedures.

Blinker can only check the stack when executing overlaid code. However, some applications making use of third party libraries linked into the root, especially in a network environment, may require an increase in the PROCEDURE DEPTH. Symptoms indicating this include erratic hanging of the machine, spurious characters on the screen and other fatal errors which do not happen regularly.

See Also:

BLINKER SEGMENT THRESHOLD   Modify the default segment packing limit

---

Purpose : **Modify the default segment packing limit**

Syntax   : **BLINKER SEGMENT THRESHOLD <nuLimit>**

Default : **65500**

Example : **# Pack segments to a limit of 16K**
BLINKER SEGMENT THRESHOLD 16384

When creating a new EXE format executable, ie DOS extended or Windows programs, Blinker packs code and data segments in order to reduce the number of segments present in the final executable. At runtime data and code are loaded and discarded as complete segments, which can impact execution speed when segment swapping occurs frequently. This command can be used to set the maximum size (in bytes) of combined or packed segments, reducing the sizes of the segments being swapped.

A decrease in the segment threshold will result in a corresponding increase in the number of segments in the EXE file. The new EXE format has a maximum limit of 254 segments, therefore, if the threshold is set too low, (causing more than 254 segments to be required), Blinker will display link time error message 1141 indicating that the new EXE segment limit has been exceeded.

Purpose : **Specify additional segment classes to be treated as CODE**

Syntax : **CODECLASS <classname> [,<classname> ...]**

Default : **None**

Example : **# Treat segments of class MYCLASS1 and MYCLASS2 as CODE**
CODECLASS MYCLASS1,MYCLASS2

This command is used to specify additional segment classes which are to be treated as CODE by Blinker. By default, all classes with a name ending in CODE are treated as CODE.

This command is useful to ensure that segments in a new .EXE file are flagged as CODE.

See Also:

Purpose : **CodeView debugger support**

Syntax : **CODEVIEW**

Default : **Disabled**

The CODEVIEW command globally and retroactively enables the processing of CodeView debugging information, regardless of the position of the CODEVIEW statement in the link script file. The CODEVIEW command is identical in effect to a CVON command at the beginning of the link script.

Developers with large overlaid programs may see some performance degradation of their overlaid code whilst running under the CodeView debugger if the CODEVIEW command has been used. If performance is an issue under these circumstances, it is recommended to use the CVON and CVOFF commands to create CodeView areas which selectively enable debugging information for specific files, modules or libraries.

The default version of CodeView information added to the .EXE file is 0 for maximum compatibility with other debuggers. Later versions may be selected using the CVVERSION command. (For more details, see the section on CVVERSION in Chapter 9.)

Note: External overlays are required in order to use this command.

See Also:

Purpose : **Locally disable the processing of CodeView information.**

Syntax : **CVOFF**

Default : **Disabled**

The CVOFF command ends a CodeView area, and disables the processing of CodeView information from the point the command is encountered in the script until the next CVON command. CodeView areas are used to limit the amount of overlaid code for which CodeView debugging is enabled.

Large overlaid applications will execute faster under the debugger if the amount of overlaid code containing debugging information is limited using CodeView areas.

See Also:

Purpose : **Locally enable the processing of CodeView information**

Syntax : **CVON**

Default : **Disabled**

The CVON command begins a CodeView area, and enables the processing of CodeView information from the point the command is encountered in the script until the next CVOFF command. The CodeView areas are used to limit the amount of overlaid code for which CodeView debugging is enabled.

Due to the method used by Blinker to support debugging of dynamically overlaid code under CodeView, large overlaid applications will execute faster under the debugger if the amount of overlaid code containing debugging information is limited using these commands.

Files, modules or libraries which are within a CodeView area will have all of their symbolic information recorded in the executable for use by the debugger. Files, modules or libraries which are not within a CodeView area will only have their public symbols and module boundaries recorded.

Example :

```
FILE a,b,c
SECTION INTO abc_xyz
BEGINAREA
   FILE d,e
   # Begin CodeView area for files f and g
   CVON
   FILE f,g
   # End CodeView area
   CVOFF
   LIB abc
   # Begin CodeView area for XYZ.LIB
   CVON
   LIB xyz
   # End CodeView area
   CVOFF
ENDAREA
LIB funclib
```

See Also:

Purpose : **Set the version of CodeView information to use**

Syntax : **CVVERSION <0 | 1 | 2 | 5>**

Default : **0**

The CVVERSION sets the version of CodeView information written to the .EXE file created by Blinker. The CodeView information version number does not refer to the version number of the CodeView debugger program, but rather the version number of the CodeView information format. At the time of writing, there are four possible versions, 0, 1, 2 or 5:

```
0  All versions of the CodeView debugger through version 2.2
1  The version of CodeView shipped with MS BASIC PDS 7.0
2  The version of CodeView shipped with MS C 6.0
5  The version of CodeView shipped with MS C/C++ 7.0, MS Visual
   C++ and MS Visual Basic for DOS 1.0
```

Blinker defaults to producing executables which use CodeView information format 0.

If you instruct Blinker to produce an executable to a lower version number than the version your CodeView debugger supports, some debugger features may not function correctly. Use the highest version number compatible with your debugger.

Note: When using CodeView information version 5, please ensure Microsoft's CVPACK utility is in the path.

See Also:

Purpose : **Specify the start of module definition commands**

Default : **None**

Syntax : **DEFBEGIN**

Example : **# Specify program name without a .DEF file**

```
DEFBEGIN
NAME TEST
DEFEND
```

This command is used to specify the start of module definition commands included in a standard freeformat link script file. This saves maintaining two separate link files for DOS extended or Windows programs. Subsequent link script file lines will be treated as module definition file commands until a corresponding DEFEND command is found.

For more details of the valid module definition commands please refer to Chapter 6, Windows Programs.

See Also:

Purpose : **Specify the end of module definition commands**

Default : **None**

Syntax : **DEFEND**

Example : **# Specify program name without a .DEF file**

```
DEFBEGIN
NAME TEST
DEFEND
```

This command is used to specify the end of module definition commands included in a standard freeformat link script file. This saves maintaining two separate link files for DOS extended or Windows programs.

For more details of the valid module definition commands please refer to Chapter 6, Windows Programs.

See Also:

Purpose : **Specify module definition command file**

Default : **None**

Syntax : **DEFFILE <filename>**

Example : **# Specify module definition file TEST**
DEFFILE TEST

This command is used to specify the name of a module definition command file in a standard freeformat link script file. A default file extension of .DEF is assumed if none is specified.

Processing of the link script file will be suspended while the module definition file specified (and any nested files therein) are processed. Processing will then continue with the next line of the link script file.

For more details of the valid module definition commands please refer to Chapter 6, Windows Programs.

See Also:

Purpose : **Specify symbols to be excluded at link time**

Syntax : **DEFINE <symbol>[, <symbol> ...]**

Example : **DEFINE Alpha, Beta**

The DEFINE command is used to 'stub out' the named symbols and point them to a dummy routine in the overlay manager. Instead of including the modules containing the named symbols in the final .EXE file, Blinker maps each DEFINEd symbol to a routine within the overlay manager which will display the runtime error 1213 if the routine is ever called. This command can be used (with care) to prevent routines which you know will never be called from being linked in.

Note that in order to prevent a module from being linked in you must DEFINE all its public symbols.

Take care to ensure that only code symbols are stubbed using DEFINE. If DEFINEd data items are written to, the overlay manager may be corrupted.

Please note that when MIXCASE is mentioned prior to DEFINE commands the symbols named will be processed as case sensitive.

See Also:

Purpose : **Use default library search records**

Syntax : **DEFLIB**

Default : **Enabled**

The DEFLIB command is used to specify that Blinker should locate and link libraries which are specified within an object module. Some compilers insert information into each generated object module, specifying the names of libraries which the linker may need at link time. This command is enabled by default.

See Also:

Purpose : **Use Microsoft DOS segment ordering**

Syntax : **DOSSEG**

Default : **Disabled**

The DOSSEG command is used to specify that Blinker should arrange the segments making up the program in conformance with the segment ordering scheme used by the Microsoft high level language compilers.

Most compilers which require this segment ordering include a DOSSEG request in each .OBJ created or in the language library, so this command is not needed under normal circumstances.

Watcom C/C++ requires an explicit DOSSEG command.

Purpose : **ECHO script commands or text to the screen**

Syntax : **ECHO [ON | OFF |] [<cString>]**

Default : **OFF**

ECHO <cString> causes <cString> to be echoed to the screen regardless of, and without affecting, the ECHO status.

ECHO ON <cString> turns on echoing of the script file to the screen, with the first line displayed being <cString> if it is specified.

ECHO OFF turns off echoing to the screen of the script file and ignores the rest of the line.

ENDAREA **Specify the end of an overlay area**

Purpose : **Specify the end of an overlay area**

Syntax : **ENDAREA**

The ENDAREA command is used to specify to the linker the end of an overlay area, as described in the BEGINAREA section.

See Also:

EXTDICTIONARY **Enable use of library extended dictionaries if they exist**

Purpose : **Enable use of library extended dictionaries if they exist**

Syntax : **EXTDICTIONARY**

Default : **Enabled**

This command specifies that Blinker should use the extended dictionary that is appended to library files by some librarian utility programs. The extended dictionary is a cross reference of the library contents, and is used by Blinker to speed up the processing of libraries. Not all librarians provide an extended dictionary - in this case Blinker will process the library slightly slower than if it were present.

See Also:

Purpose : **Limit the amount of memory allocated to the program**

Syntax : **EXTRAMEM nuSize**

Default : **1 MB - (physical image size)**

The EXTRAMEM command is used to limit the amount of extra memory DOS will allocate to a program. The parameter is the size, as a decimal number of paragraphs beyond the image size, of the largest memory block into which DOS will load the program. This is equivalent to the /CPARM option of MSLink.

This command is used, amongst other things, to reduce the size of the near heap in Medium and Large model programs.

There are three figures associated with the memory requirements of a DOS executable file:

a) The physical image size of the root portion of the executable.
b) The additional amount of memory required for uninitialised data.
c) Additional memory requested by the program. This value is normally set to 1MB, causing DOS to allocate the largest available contiguous memory block to the program at load time.

The figure represented by (a+b) is known as the load size, and is the amount of memory that DOS requires to load the program into memory.

The EXTRAMEM command specifies a number of paragraphs to be allocated to the program in ADDITION to the physical image size, and must be at least as large as (b). If the number of paragraphs specified is LESS than (b), Blinker will automatically increase the EXTRAMEM figure to the minimum value.

See Also:

Purpose : **Specify object file(s)**

Syntax : **FILE [d:][path]<filename> [, [d:][path]<filename> ...]**

Example : **FILE Myapp1, D:\APPS\OBJ\Myapp2**

The FILE command is used to specify to the linker the names of one or more .OBJ files to be included in the output file at that point. The default file extension of .OBJ need not be specified. The first file specified will generally be the program's main module.

Several filenames may be specified within a single FILE command, separated by commas. Blinker will interpret a carriage return / linefeed combination as the end of a FILE command, therefore the FILE command must be restated at the beginning of each line.

All files listed in FILE commands are included in the linked program, whether they are required or not. To include only the files needed by the specific program, a library (.LIB) file should be created using a librarian such as Microsoft LIB.EXE. This library can then be linked into the program using the LIBRARY command, and only the required .OBJ files from the .LIB file will be linked to the program.

Blinker looks for each file first in the current directory and then in each directory path specified in the OBJ environment variable in the order they appear. Alternatively, a full drive and directory path may be specified with any or each of the file names.

The FILE command can be used on a library file to force all the modules within the library to be included in the .EXE. In this case the .LIB extension and full path will probably need to be specified, e.g. FILE path\CLD.LIB.

See Also:

Purpose : **Specify one or more program libraries**

Syntax : **LIBRARY [d:][path]<libname>**
[, [d:][path]<libname> ...]

Default : **Compiler specified default libraries**

Example : **LIB C:\LIBS\Mylib1, Mylib2, D:\OTHER\Mylib3**

The LIBRARY command is used to specify to the linker the names of one or more program library files required by the program. The default file extension of .LIB is assumed, and need not be specified. If more than one library is specified in a single LIBRARY command, each library name should be separated by a comma.

Library files are processed differently from FILE commands in that a LIBRARY command causes only modules which are required by the program to be included. i.e. modules are included from a library if a public definition in the library module resolves an external reference. Object files named in FILE commands are included regardless of whether they are required or not.

The LIBRARY command may be used for both root and overlaid libraries. If the library is to be overlaid, the LIBRARY command should be placed within a BEGINAREA / ENDAREA sequence.

See Also:

MAP **Request a segment map of the executable**

Purpose : **Request a segment map of the executable**

Syntax : **MAP [=[d:][path]<filename>] [S][,A]]**

Default : **Disabled**

Example : **# Generate a map of all public symbols**
MAP=C:\TEMP\Mymap A

The MAP command is used to specify to the linker that a map file of the linked .EXE file is required. The map file defaults to the same name as the .EXE file with an extension of .MAP, or the name can be specified in the MAP command. The map file can be listed with varying levels of detail, as required.

The two options to the command are as follows:

S - All segments - lists the address, length and name of each segment in the executable image. This is the default option, if none is specified.

A - Public symbols in address order - lists the public symbols and their memory addresses. The A option can be used to produce a map file which can be used for debugging C and Assembler programs using debuggers which do not support CodeView information. The Turbo Debugger is not included in this category since the Borland TDCONVRT utility allows the Turbo Debugger to make full use of CodeView format .EXE files.

For a discussion of the output generated by the MAP command see Appendix A.

See Also:

Purpose : **Limit the amount of memory allocated to the program**

Syntax : **MEMORY <nuLimit>**

Default : **1 MB**

The MEMORY command is used to set the size, as a decimal number of paragraphs, of the largest memory block into which DOS will load the program.

This command is used, amongst other things, to reduce the size of the near heap in Medium and Large model programs.

There are three figures associated with the memory requirements of a DOS executable file:

a) The physical image size of the root portion of the executable.
b) The additional amount of memory required for uninitialised data.
c) Additional memory requested by the program. This value is normally set to 1MB, causing DOS to allocate the largest available contiguous memory block to the program at load time.

The figure represented by (a+b) is known as the load size, and is the amount of memory that DOS requires to load the program into memory.

The MEMORY command specifies the total number of paragraphs to be allocated to the program (a+b+c), and must be at least as large as (a+b). If the amount of memory specified is LESS than (a+b), Blinker will automatically increase the MEMORY figure to the minimum value.

See Also:

Purpose : **Specify case sensitivity**

Syntax : **MIXCASE [EXPORT]**

Default : **Disabled**

The MIXCASE command causes Blinker to treat all publics and externals as case sensitive at link time, rather than converting them to UPPERCASE. This command is of primary interest to developers using C, who may require case sensitivity.

By default all export definition names in module definition files are converted to uppercase when they are processed. When MIXCASE EXPORT is specified in the link script file the export names are treated as case sensitive.

See Also:

Purpose : **Specify placement of individual library modules**

Syntax : **MODULE <module> [,<module> ...]**
[FROM <libname>]

Example : **# Move the module COLOR to the current location**
MODULE COLOR

The MODULE command allows individual modules to be forced into the root or the overlay area regardless of whether the originating libraries or object modules are specified within an overlay area or not.

The module name refers to an object module that exists within a library which has been specified elsewhere in the link script with a LIB or SEARCH command. Blinker's VERBOSE command output lists all modules that are processed at link time.

The position of the module command within the link script determines the placement of the named module within the executable - if the MODULE command is in the root the named modules will be forced into the root; if it is in the overlay area, the named modules will be overlaid. Multiple modules can be specified on the same command line separated by ','.

The library and module names should be specified without path names; if included, they will be ignored. The VERBOSE command can be used to determine the correct module name to use, since it displays each module name as it is processed from the library or object module. This is the same name as used by the Microsoft librarian (LIB.EXE).

MODULE commands are case insensitive, (upper or lower case will be treated the same) unless the MIXCASE command has been specified in the script file before the MODULE command.

The MODULE command has a second mode of operation, whereby a source library is specified using the FROM <libname> option. This allows modules from specific libraries to be used in preference to others, and at the same time performs the normal MODULE function of changing the placement of routines in the root or overlay area. This is useful to resolve a conflict between two libraries which contain modules with the same name.

It is advisable to use the FROM option only when necessary, since it modifies the normal linking sequence.

MODULE DBEDIT, MEMOEDIT FROM EXTEND

This command forces the modules DBEDIT and MEMOEDIT from EXTEND to be used in preference to modules of the same name which may have been found previously in other libraries or files, AND moves those modules to the current location in the script file.

MODULE MEMOEDIT FROM NUL

This command forces the module MEMOEDIT not to be linked at all, since Blinker will be looking for a library called NUL, which cannot exist. If, however, something in MEMOEDIT is explicitly referenced by another module, this will cause an unresolved external. To override this, use the DEFINE command rather than the MODULE command, and DEFINE the public symbols within the module which are unresolved.

See Also:

Purpose : **Create a worst case overlay situation**

Syntax : **MURPHY**

Default : **Disabled**

The MURPHY command causes the overlay manager to attempt to force any inadvertent overlay errors (programming errors) to manifest themselves during testing or debugging.

When MURPHY is enabled, the overlay manager will only allow a single overlay to be in memory at any given time, allowing the developer to detect 'badly behaved' dependencies between overlays and correct them.

The overlay manager also fills all unused space in the overlay pool with INT 3 instructions, with a far jump to an error reporting routine within the overlay manager being placed at the very end of the overlay area.

When the program is run under a debugger, and an erroneous jump to an invalid (i.e. non-vectored) address within the overlay area occurs, the INT 3 instruction will cause the debugger to breakpoint at the offending address.

If the program is not being executed under a debugger, the processor will execute all the INT 3 instructions and the far jump to the error routine in the overlay manager, at which point the overlay manager will display a message and terminate the program.

The use of the MURPHY command may slow down the execution of the program significantly.

Purpose : **Suppress the beep upon completion of link**

Syntax : **NOBELL**

Default : **Disabled**

The NOBELL command is used to specify to the linker that no beep is to sound when the link has completed.

Purpose : **Ignore default library search records**

Syntax : **NODEFLIB**

Default : **Disabled**

The NODEFLIB command is used to specify that Blinker should ignore default library search records which are encountered within object modules. Some compilers insert information into each generated object module, specifying the names of libraries which the linker may require. Blinker defaults to using these records unless NODEFLIB is specified.

See Also:

NOEXTDICTIONARY **Disable processing of extended library dictionaries**

Purpose : **Disable processing of extended library dictionaries**

Syntax : **NOEXTDICTIONARY**

Default : **Disabled**

This command specifies that Blinker should ignore the extended dictionary which is appended to library files by some librarian utility programs. The extended dictionary is a cross reference of the library contents, and is used by Blinker to speed up the processing of libraries.

If use of the extended dictionary is disabled using this command, Blinker will process the library slightly slower. Some librarians have been known to create invalid extended dictionaries, which can lead to erratic results at link time - this command will ensure that such libraries will be processed correctly.

This command applies globally to all libraries processed during the link.

See Also:

Purpose : **Disable packing of NEW EXE code segments**

Syntax : **NOPACKCODE**

Default : **Disabled**

This command specifies that Blinker should not pack code segments when creating NEW exe format executables (Windows and DOS extended programs.)

Blinker defaults to packing code segments when creating NEW EXE format programs, as the NEW EXE format allows for only a limited number of segments (254).

See Also:

NOPACKDATA **Disable packing of NEW EXE data segments**

Purpose : **Disable packing of NEW EXE data segments**

Syntax : **NOPACKDATA**

Default : **Enabled**

This command specifies that Blinker should not pack data segments when creating NEW exe format executables (Windows and DOS extended programs.)

Blinker defaults to not packing data segments when creating NEW EXE format programs.

See Also:

Purpose : **Disable processing of library table of contents**

Syntax : **NOTABLEOFCONTENTS**

Default : **Disabled**

This command forces Blinker to SEARCH all libraries, i.e. ignore all the library table of contents. In certain unusual circumstances this command can be used to emulate the order in which Borland's TLINK linker processes modules within a library.

This command applies globally to all libraries processed during the link.

Purpose : **Specify the width of symbol names in the map file.**

Syntax : **NWIDTH <nuWidth>**

Default : **23**

This command specifies the maximum length of a symbol name which will be output to the map file. Symbol names which are longer than the NWIDTH value will be truncated.

See Also:

Purpose : **Specify the name of the output executable file**

Syntax : **OUTPUT [drive:][path]<filename>**

Default : **<FirstObj>.EXE**

Example : **# Create a linked file called OUT.EXE**
OUTPUT OUT.EXE

The OUTPUT command is used to specify to the linker the drive, path and filename of the output executable file to be created by the linker. If no OUTPUT command is used then the linker will use the name and path of the first object file appearing on a FILE command in the script file (or on the command line).

The OUTPUT file name extension does not have to be .EXE, but if no file extension is specified, a default of .EXE is used.

See Also:

Purpose : **Specify which segment classes are to be overlaid**

Syntax : **OVERLAY <classname> [,<classname> ...]**

Default : **The class of the first segment encountered in the link process.**

Example : **OVERLAY CODE**

The OVERLAY command is used to specify to the linker which segment class or classes from the object files are to be placed in the overlay sections.

The overlay class defaults to the first segment class processed during linking and is displayed to the screen at the end of the first pass when VERBOSE mode is selected. It should be unnecessary to use this command for high level language compilers as they automatically place all code segments at the start of the .OBJ.

Assembler programmers have complete control over the segment classes and their order in the .OBJ files, so they may have need of the OVERLAY command to force overlaying of code not in segments of class CODE.

Blinker only supports dynamic overlaying of program code, so use this command with care, and do not attempt to overlay data segments.

See Also:

PACKCODE **Enable packing of NEW EXE code segments**

Purpose : **Enable packing of NEW EXE code segments**

Syntax : **PACKCODE [nuThreshold]**

Default : **Enabled (threshold=65500)**

This command specifies that Blinker should combine code segments when creating NEW exe format executables (Windows and DOS extended programs). The optional parameter is the maximum size for each segment before starting a new one. Code packing is enabled by default with a threshold of 65500 bytes, which is also the default if no parameter is specified.

See Also:

PACKDATA **Disable packing of NEW EXE data segments**

Purpose : **Disable packing of NEW EXE data segments**

Syntax : **PACKDATA [nuThreshold]**

Default : **Disabled**

This command specifies that Blinker should pack data segments when creating NEW exe format executables (Windows and DOS Extended programs). The optional parameter is the maximum size for each segment before starting a new one. Data packing is disabled by default except when symbol table compression is enabled for CA-Clipper 5.x programs.

The default threshold value, unless otherwise specified is 65535 bytes.

See Also:

READONLY **Specify executable file will be set to read only status.**

Purpose : **Specify executable file will be set to read only status.**

Syntax : **READONLY**

Default : **Disabled**

The READONLY command causes Blinker to create the .EXE file with a read only file attribute. This command will be of primary interest to developers linking and testing a program over a network, as it prevents sharing violations when executing the program from several workstations.

Note that the read only attribute is lost if the file is copied.

See Also:

Purpose : **Prioritize all symbols in the specified libraries.**

Syntax : **SEARCH [d:][path]<libname> [, [d:][path]<libname> ...]**

Default : **Disabled**

Example : **SEARCH C:\LIBS\MYLIB1, D:\ABC\LIB2**

The SEARCH command is an alternative form of the LIBRARY command, but operates slightly differently.

During the normal linking process, Blinker will make multiple passes over a given library until it is unable to resolve any further external references, at which point processing of the next library begins. If, after all libraries have been processed, there are still unresolved externals, Blinker will make another pass over the libraries and so on.

When Blinker encounters a SEARCH command, it loads all the symbol names from the named library into a table in memory. This has the effect of causing any previous unresolved external references, and any future external references encountered to be resolved from the SEARCHed library (if they are defined there) in preference to a subsequent library.

Use of the SEARCH command should be limited to those cases where it is required, as it slows the linking process, and uses large amounts of memory at link time.

See Also:

Purpose : **Specify files to be placed in an external overlay**

Syntax : **SECTION INTO <ovlname> [FILE [d:][path]<filename>**
[, [d:][path]<filename>...]]

Default : **Disabled**

Example :
```
FILE MAIN
BEGINAREA
SECTION INTO OVL1
FILE ABC, DEF, GHI # Placed in OVL1
SECTION INTO OVL2
FILE JKL, MNO, PQR # Placed in OVL2
ENDAREA
```

The SECTION INTO command is used to specify the name of an external overlay file for the overlays. The file may have any extension except .EXE, and if no extension is specified then .OVL is assumed. The first SECTION INTO command must precede the first overlaid FILE command, otherwise an internal overlay within the OUTPUT file will be created and all SECTION INTO commands will be ignored. Note that this command does NOT cause overlays to be created, it merely causes overlays (if any) to be stored external to the EXE file (in OVL files). The BEGINAREA and ENDAREA commands are used to create overlays.

Once an overlay file has been specified using the SECTION INTO command, all overlaid code will be placed in the same file until a SECTION INTO command with a different file name is found. At this time a new overlay file will be created. Output may be spread between the overlay files by the use of multiple SECTION INTO commands.

Once an internal overlay has been created, all SECTION INTOs will be ignored and all overlaid procedures will be stored internally. Once an external overlay has been created it is not possible to create an internal overlay. Blinker has a limit of 15 external overlays in this version.

At program runtime, one file handle is allocated to each overlay file, so it may be necessary to increase the number of file handles available to the program.

See Also:

STACK **Specify stack size of .EXE file**

Purpose : **Specify stack size of .EXE file**

Syntax : **STACK <nuSize>**

Default : **Determined by compiler**

Example : **STACK 6148 # Set stack size to 6148 bytes**

The STACK command is used to specify to Blinker in DECIMAL the stack size in bytes required by the program .EXE file. Please note that the stack size is specified in DECIMAL, whereas the Microsoft Link stack size is specified in HEX. Therefore, a stack size of 1000 with Microsoft Link would directly equate to a stack size of 4096 with Blinker.

Notes for CA-Clipper:

For CA-Clipper developers, a preferable way of increasing the stack size and therefore the maximum depth of procedure calls is to use the BLINKER PROCEDURE DEPTH command.

BLINKER's default stack size when linking CA-Clipper programs is 5120 bytes, which corresponds to a PROCEDURE DEPTH of approximately 50.

See Also:

UPPERCASE **Convert all symbols to uppercase**

Purpose : **Convert all symbols to uppercase**

Syntax : **UPPERCASE**

Default : **Enabled**

The UPPERCASE command is used to specify to the linker that all symbols should be converted to upper case before being added to the symbol table. Blinker defaults to uppercase.

See Also:

Purpose : **Display status information during linking**

Syntax : **VERBOSE [0|1|2]**

Default : **Disabled**

Example : **# Create VERBOSE output and redirect to TEST.MSG**
BLINKER @TEST VERBOSE >TEST.MSG

The VERBOSE command is used to specify to the linker that it is to display information regarding the current operation being performed. This command will slow down the linking process significantly, but provides the developer with a means of isolating the cause of a link time error.

When VERBOSE is specified, Blinker displays each object module and the file containing it as it is processed. This output may be redirected to a file using standard DOS redirection if required. VERBOSE's numerical parameter indicates the desired level of output detail:

0 disables VERBOSE output
1 outputs filenames while processing
2 outputs filenames and module names (default if no number specified)

For an example of output generated by the VERBOSE command, see Appendix A.

See Also:

[Next Entry>>]

WORKFILE **Specify drive, path, and file name of temporary work file**

Purpose : **Specify drive, path, and file name of temporary work file**

Syntax : **WORKFILE [d:][path]<filename>**

Example : **WORKFILE C:\TEMP\Temp**

The WORKFILE command is used to specify to the linker the drive, path and file name of the temporary work file to use at link time if one is required. By default, Blinker will use either XMS (2.0 or higher) or EMS (3.2 or higher) memory in preference to a workfile, when available.

If no WORKFILE command is specified and Blinker needs to swap to disk during a link, it will create a unique file in the directory specified by the TEMP environment variable if one exists, or the current directory if it does not exist.

[^^Up^^]

[<<Previous Entry]

BLINKER Programming Functions

Function Overview

Blinker features a number of functions which provide the program developer with increased control over the runtime overlay environment and additional debugging information. All Blinker functions used by your program are automatically linked without the need to specify any additional libraries. Blinker functions can only be used in Blinker linked programs.

All the functions which produce output take a file handle as an optional parameter, and all output from the function will be sent to that file handle. The handle is not validated, and should be either 1, to send output to the screen, 2 to send output to the standard error device, 4 to send output to the printer, or the handle of a file opened by the program.

The default handle may be set at link time using the BLINKER DEBUG HANDLE command or in the program using the Debug Handle function (BLIDBGHAN()) detailed below. If no handle is specified then the output is sent to DOS file handle 1 (the screen).

Naming Conventions

Assembler programmers note: when calling from assembler you should prepend an underbar character to the function names in order to comply with C naming conventions.

BASIC programmers note: Blinker functions called from BASIC which receive string parameters or return string values have a different function name - the letters 'BAS' are appended to the normal nine character function name. For example, the BASIC version of BLIDEMDTE() is called BLIDEMDTEBAS().

Notational Conventions

Many of the functions provided by Blinker can be accessed from several languages, using the standard C calling conventions.

Blinker provides specific interfaces for the C language, Microsoft Basic PDS & QuickBasic, and for CA-Clipper Summer '87 and 5.01. If you are using a language not mentioned here, it is likely that your compiler can be directed to use the C calling convention to access Blinker functions. Consult your compiler documentation for further information on mixed language programming.

The tables on the following pages detail the notational convention used to describe the types of parameters and return values, and the appropriate data types for each calling convention.

Each parameter, (or return value) is preceded by a two letter prefix denoting the type of the value, which also implies the way the parameter is passed to the function.

Parameter type conventions

The following tables show the data types used as parameters and return values from all Blinker functions. The prefixes have the following meanings:

```
ni numeric     signed integer    (signed word)
nu numeric     unsigned integer  (unsigned word)
nl numeric     long integer      (signed dword)
c  character   string            (byte pointer)
l  logical     (boolean)         (word)
```

For each language listed in the table (CA-Clipper, Basic, and C) the language declaration for each type is given (no type declaration is needed

in the CA-Clipper language, but the passed parameters must be of the correct type).

Logical (or boolean) values are only directly supported for CA-Clipper. All other languages should use the numeric integer type, passing a zero value to indicate false, and a non zero value to indicate true. Return values from the functions use the same convention.

For languages not listed, use the data types which match the C types.

Calling conventions

The Blinker functions are provided with interfaces to most languages, and these are defined in the language include files BLINKER.BI, BLINKER.H, BLINKER.CH, BLINKER.FI, BLINKER.HDR, BLINKER.DEF and BLINKER.INC,. These files can be found in the BAS, C, CLP, FOR, FRC, MOD and PAS subdirectories respectively of the Blinker installation directory.

The CA-Clipper interface uses the CA-Clipper extend system, and requires no special treatment as far as the CA-Clipper programmer is concerned.

The BASIC interface is identical to the C interface for simple data types, but strings require special treatment. For this reason, any of the Blinker functions which pass or return string values have the suffix 'BAS' appended to the normal function name.

The C interface uses the following calling conventions:

- Simple data types are passed on the stack
- Strings are passed by far reference
- Parameters are passed in reverse order, ie the last parameter is pushed onto the stack first.

which are the same as standard C calling conventions.

Languages for which no interface is provided should use the C interface.

Function : **Get / Set the Blinker debug handle**

Syntax : **nuValue = BLIDBGHAN([nuhandle])**

Return : **Previous debug Handle**

This function is used to get or set the Blinker overlay manager debug handle. It returns the previous debug handle, and takes as an optional parameter the new handle it should be set to.

The handle defaults to 1 for the screen unless it is overridden at link time using the BLINKER DEBUG HANDLE command or at run time using the /DHn setting in the BLINKER environment variable.

The new handle is not validated and may be either 1 to send output to the screen, 2 to send output to the standard error device, 4 to send output to the printer, or the DOS handle of a file opened by the program with a DOS open file call.

This handle is used as the default output handle for a number of the other Blinker debugging functions, so may be set once at the start of the program rather than passing it as a parameter to each function individually.

Example:

```
* Get the current debug handle
*
? "The debug handle is currently", BLIDBGHAN()
* Set the current debug handle to the printer
BLIDBGHAN(4)
```

See Also:

BLIDEMDTE() / BLIDEMDTEBAS() Get demonstration date

Function : **Get demonstration date**

Syntax   : **cValue = BLIDEMDTE()**

Return   : **Demonstration date as a string**

This function returns the demonstration date limit of the program if it was set at link time, and returns it as a string in the form YYYYMMDD.

A null string will be returned if the demonstration date limit has not been set.

Example:

```
   * Display the expiration date if there is one
   *
   TMPDEMDTE = BLIDEMDTE()    && Get the date limit
   IF LEN(TMPDEMDTE) > 0
      ? "The expiration date of this program is "
      ?? TMPDEMDTE
   ENDIF
```

See Also:

Function : **Get / Set demonstration minutes**

Syntax : **nuValue = BLIDEMMIN([nuLimit])**

Return : **Demonstration minutes limit**

This function returns the demonstration time limit of the program if it was set at link time, and returns an integer number of minutes between 0 (not set) and 65535.

It may also be used to extend the limit to allow the current task to finish by passing the new limit as a parameter to the function. Since this is the new limit it will always be greater than the value returned.

Example:

```
* Get the time limit and extend it by 2 minutes
*
TMPDEMMIN = BLIDEMMIN() && Get the time limit
IF TMPDEMMIN > 0
   ?"The time limit is", TMPDEMMIN, "minutes"
   ?"It will be extended by 2 minutes"
   BLIDEMMIN(TMPDEMMIN + 2)
ENDIF
```

See Also:

Function : **Display CA-Clipper Summer '87 free pool fragmentation**

Syntax   : **BLIDISFRG([nuHandle])**

Return   : **None**

This function is used to display a summary of CA-Clipper Summer '87's dynamic memory to show the degree of fragmentation, and takes as an optional parameter the output destination file handle. Each character position represents up to 512 bytes, and adjacent equal characters represent contiguous blocks larger than 512 bytes.

Two memory areas are mapped by this function; the largest area is the far heap, often referred to as the FREE POOL, where most of CA-Clipper's memory allocations are made, and the near heap, a much smaller block of contiguous memory (maximum size 64kb), which CA-Clipper will use if an allocation cannot be satisfied from the free pool.

The _ and _ characters are used to indicate blocks of free memory which are unused by CA-Clipper, and the two graphics characters _ and _ are used to indicate blocks of memory which are in use, i.e. have been allocated. The character . is used to indicate the point at which the next memory allocation will occur. A key is shown below the memory map.The reason for having two different characters for free blocks and two different characters for used blocks is to show where a free or used area is made up of more than one block of memory. If only one character was used for each then the display would not show the full extent of fragmentation of memory into very small blocks.

The relative sizes of blocks are shown by identical repeated characters, since a single character indicates a memory block of less than 512 bytes, and a row of four blocks indicates a memory block of approximately 2kb. The relative fragmentation is shown by the proportion of free blocks to used blocks and by the alternation between free and used blocks throughout the display.

Once free areas are separated by used areas the free areas become virtually unusable except for the storage of small temporary values, which is why MEMORY(0) gradually deteriorates and operations which require larger blocks of memory such as MEMOEDIT() or indexing will then fail.

The BLIDISFRG() function may be used during the execution of an application to monitor the fragmentation of memory as execution continues. It may also be used to view the improvements gained using the BLINKER MEMORY PACK command and the BLIMEMPAK() function.

Note that the CA-Clipper Summer'87 free pool is allocated from DOS on an 'as-needed' basis. Unless MEMORY(0), BLIMEMPAK(-1) or BLIMEMBLK() are executed (which cause CA-Clipper to allocate all available memory to the free pool from DOS), this function will reflect only the currently allocated free pool.

Example:

```
* Display the memory fragmentation
*
BLIDISFRG(4)      && Send to the printer
```

See Also:

BLIERRNUM() **Get Blinker error number**

Function : **Get Blinker error number**

Syntax : **nuValue = BLIERRNUM()**

Return : **Blinker Error Number**

This function returns the Blinker runtime error number, and is for use in the Blinker error handler BLERRPRG. In normal operation the value returned is zero, meaning no error.

For details of modifying the Blinker error handler see Chapter 7, Special Features

For details of the Blinker runtime errors see Chapter 11, Errors and Error Messages.

Example:

```
* Get the Blinker error number
*
? "The BLINKER error number is", BLIERRNUM()
```

See Also:

[<<Previous Entry]

BLINKER Script Commands Overview

This section is intended as a reference for all the Blinker script file commands, but a thorough understanding of these commands should be obtained before attempting to create a program or to convert an existing program for use with Blinker.

Script file(s) are specified on the command line and default to an extension of .LNK. They may be nested, and more than one may be used on the command line, e.g.:

BLINKER @Defaults @Other LIB Mylib VERBOSE

This command line instructs Blinker to read the script file DEFAULTS.LNK followed by the script file OTHER.LNK. It also instructs Blinker to use the library MYLIB.LIB and to display a VERBOSE listing of the .OBJ files being processed.

The full names of the commands or any unique abbreviation of the words which comprise the commands may be used, although it is recommended that at least three letters be used for clarity. For example, the command

BLINKER OVERLAY OPSIZE 40

can be abbreviated to BLI OVE OPS 40.

BLIERRPRM() / BLIERRPRMBAS() Get Blinker error parameter

Function : **Get Blinker error parameter**

Syntax   : **cValue = BLIERRPRM()**

Return   : **String containing name of file related to the Blinker error**

Some of the Blinker runtime errors refer to a particular file name, and this function returns this file name for use in the error message.

For details of modifying the Blinker error handler see Chapter 7, Special Features

For details of the Blinker runtime errors see chapter 11 Errors and Error Messages.

Example:

```
* Get the Blinker error parameter
*
? "The file causing the BLINKER error is", BLIERRPRM()
```

See Also:

BLILSTFRG() List CA-Clipper Summer '87 memory blocks

Function : **List CA-Clipper Summer '87 memory blocks**

Syntax : **BLILSTFRG([nuHandle])**

Return : **None**

This function is used to list each block of CA-Clipper's dynamic memory to show the degree of fragmentation. Each block is listed on a separate line with its address in memory, its size and whether it is in use or not. The function takes as an optional parameter the output destination file handle.

Note that the CA-Clipper free pool is allocated from DOS on an 'as-needed' basis, and unless MEMORY(0), BLIMEMPAK(-1) or BLIMEMBLK() are executed (which cause CA-Clipper to allocate all available memory to the free pool from DOS), this function will reflect only the currently allocated free pool.

Example:

```
* List the Fragmentation
*
BLILSTFRG(4)   && Send to the printer
```

An example of the output:

```
Address      Size     In Use
8198:0000    00018    Y
8198:0014    00184    N
8198:00EA    00058    Y
8198:0120    00920    N
```

See Also:

BLIMEMAVL()   **Get CA-Clipper total free dynamic memory**

Function : **Get CA-Clipper total free dynamic memory**

Syntax   : **nlValue = BLIMEMAVL()**

Return   : **Total available memory in bytes.**

This function returns the total dynamic memory available in bytes in the CA-Clipper free pool. Dynamic memory is the memory used by CA-Clipper to store all temporary values of variables and buffers while an application is running.

Note that the CA-Clipper Summer '87 free pool is allocated from DOS on an 'as-needed' basis, and unless MEMORY(0), BLIMEMPAK(-1) or BLIMEMBLK() are executed (which cause CA-Clipper to allocate all available memory to the free pool from DOS), the value returned by this function will reflect only the currently allocated free pool.

Example:

```
* Get the total available memory
*
? "There are", BLIMEMAVL()
?? " bytes available in CA-Clipper's free pool"
```

See Also:

Function : **Get CA-Clipper Summer '87 free memory down to blocksize**

Syntax : **nuValue = BLIMEMBLK([nuBlocksize])**

Return : **Free memory in kb of blocks that are >= [nuBlocksize]**

This function returns the total dynamic memory available in kilobytes of blocks at least as large as the block specified in kb as a parameter. Dynamic memory is the memory used by CA-Clipper to store all temporary values of variables and buffers while an application is running.

If no parameter is specified a default block size of 4kb is used, which simulates the value returned by a call to CA-Clipper's MEMORY(0). In all cases the function will emulate MEMORY(0) and attempt to allocate blocks of size 63 Kb, then 48 Kb, then 32 Kb, then 16 Kb, then 8, 4, 2 and 1Kb until no block larger or equal to the parameter or default can be allocated. The value returned will be the total size of the blocks successfully allocated.

BLIMEMBLK() executes many times faster than MEMORY(0), so may be used to monitor memory in places where MEMORY(0) is found to slow the application too much, for example during indexing or profiling.

Example:

```
* Get the available memory in minimum of 8 Kb blocks
*
? "Memory available in at least 8Kb blocks is "
?? BLIMEMBLK(8), "Kb"
```

BLIMEMMAX() **Get largest free CA-Clipper memory block**

Function : **Get largest free CA-Clipper memory block**

Syntax : **nlValue = BLIMEMMAX()**

Return : **Size of largest free memory block in bytes.**

This function returns the size in bytes of the largest available block of dynamic memory. Dynamic memory is the memory used by CA-Clipper to store all temporary values of variables and buffers while an application is running.

This can be used to prevent the failure of memory intensive operations such as MEMOEDIT() which require large blocks of contiguous memory for successful operation.

Example:

```
* Get the size of the largest available block
*
? "The largest available block of memory is"
?? BLIMEMMAX(), "bytes"
```

See Also:

Function : **Pack CA-Clipper Summer '87 memory pool.**

Syntax : **nuValue = BLIMEMPAK([nuFrequency | -1])**

Return : **Memory Pack Frequency**

This function is used to get or set the frequency of procedure calls with which the Blinker overlay manager packs free dynamic memory to reduce fragmentation, returns the previous pack frequency, and takes an optional parameter of the new frequency to set it to.

Blinker defaults to maximum speed and a setting of zero for the frequency, which means it performs no automatic packing of free dynamic memory to prevent fragmentation. Blinker may be told to automatically pack memory after every n procedure calls, either at link time using the BLINKER MEMORY PACK command (see chapter 9 for further information) or at application runtime using the BLIMEMPAK() function.

The effect of this function depends on the value of the parameter passed, but it always returns the current pack frequency, i.e. the Blinker overlay manager will pack free dynamic memory every time this number of procedures has been called.

If no parameter is specified then no action will be taken by the function.

If a parameter value of -1 is passed, the overlay manager performs a memory pack immediately and leaves the pack frequency unchanged. This is particularly useful following a call to BLIOVLCLR() to clear inactive overlays before DBEDIT, MEMOEDIT, reindexing and other memory intensive routines.

A parameter of 0 switches off the memory packing altogether, and any other value sets the packing frequency to that value.

If automatic memory packing is not enabled using the BLINKER MEMORY PACK command at link time, then performing a BLIMEMPAK(-1) at the start of the program can often make a significant difference to memory fragmentation. The reason for this is that even by the time the first line of the CA-Clipper program has been reached a large number of dynamic memory allocations and releases have already been performed, so memory can already be partially fragmented.

Example:

```
* Get the BLINKER memory pack frequency
*
? "Packing currently occurs after every "
?? BLIMEMPAK(), "calls"
* Set to every 10 procedure calls
*
BLIMEMPAK(10)
```

See Also:

Function : **Get the size of the CA-Clipper Summer '87 dynamic memory area**

Syntax : **nlValue = BLIMEMSIZ()**

Return : **Total size of the free pool in bytes**

This function returns the total size in bytes of the dynamic memory area. Dynamic memory is the memory used by CA-Clipper to store all temporary values of variables and buffers while an application is running.

Note that the CA-Clipper Summer '87 free pool is allocated from DOS on an 'as-needed' basis, and unless MEMORY(0), BLIMEMPAK(-1) or BLIMEMBLK() are executed (which cause CA-Clipper to allocate all available memory to the free pool from DOS), the value returned by this function will reflect only the currently allocated free pool.

Example:

```
* Get the total free pool size
*
? "The total size of CA-Clipper's free pool is"
?? BLIMEMSIZ(), "bytes"
```

See Also:

Function : **Get the amount of used memory in the CA-Clipper Summer '87 memory area**

Syntax : **nlValue = BLIMEMUSE()**

Return : **Sum of the sizes of all the memory blocks marked as used.**

This function returns the size in bytes of all the dynamic memory currently in use. Dynamic memory is the memory used by CA-Clipper to store all temporary values of variables and buffers while an application is running.

Note that the CA-Clipper Summer '87 free pool is allocated from DOS on an 'as-needed' basis, and unless MEMORY(0), BLIMEMPAK(-1) or BLIMEMBLK() are executed (which cause CA-Clipper to allocate all available memory to the free pool from DOS), the value returned by this function will reflect only the currently allocated free pool.

Example:

```
* Get the amount of dynamic memory in use
*
? BLIMEMUSE(), "bytes of dynamic memory are "
?? "currently in use"
```

See Also:

Function : **Get program status information**

Syntax : **nlvalue = BLIMGRSTS(niOption)**

Return : **Various**

BLIMGRSTS() is a multi-purpose function which can be used to obtain status information concerning the currently executing program and environment.

This function can be called from most languages using the normal calling conventions for the language being used. BLIMGRSTS() takes a single parameter - an option number indicating the status information required. Information that is undefined in the current environment will be returned as zero (eg cache size in a protected mode program).

The constant parameters to BLIMGRSTS() are defined in the header files BLINKER.*. The values currently defined are :

BliHostMode	Get DOS extender host mode (DPMI/VCPI/XMS)
BliMachineMode	Get current machine mode (real, protected)
BliCacheLoc	Get location of real mode overlay cache (EMS/XMS)
BliCacheSize	Get size of overlay cache
BliOverlayLoc	Get location of overlay area
BliOverlaySize	Get size of overlay area
BliExtMemAvail	Get bytes extended memory available to the extender
BliRealMemAvail	Get bytes real memory available to the extender
BliVirMemAvail	Get bytes virtual memory available to the extender

Example :

```
* Get processor and overlay cache status
*
if (BLIMGRSTS(BliMachineMode) = BliModeReal)
   ? "Running in real mode"
   cacheloc = BLIMGRSTS(BliCacheLoc)
   do case
   case cacheloc = BliCacheNone
      ? "No cache available"
   case cacheloc = BliCacheXMS
      ? "Cache is in XMS"
   case cacheloc = BliCacheEMS
      ? "Cache is in EMS"
   endcase
else
   ? "Running in protected mode"
   hosttype = BLIMGRSTS(BliHostMode)
   ? "DOS extender host is : "
   do case
   case hosttype = BliHostDPMI
      ?? "DPMI"
   case hosttype = BliHostVCPI
      ?? "VCPI"
   case hosttype = BliHostXMS
      ?? "XMS"
   endcase
endif
```

Function : **Clear unused CA-Clipper Summer '87 overlays from memory**

Syntax : **BLIOVLCLR()**

Return : **Nothing**

This function is only effective with the BLINKER OVERLAY DYNAMIC method of overlay allocation (CA-Clipper Summer '87 only). It performs no function with the BLINKER OVERLAY FIXED method of overlay allocation.

This function is used to force the Blinker overlay manager to clear out or discard all inactive overlay procedures from the overlay pool and return the memory used by them to CA-Clipper.

This may be used in the main module of an application to ensure that CA-Clipper has the maximum amount of dynamic memory available before executing another overlay.

Once all inactive overlays are flushed, the overlay pool operating size is reset to a maximum of the default size or the total size of the active overlays. The default size when the application starts is the most recent of the link time BLINKER OVERLAY OPSIZE <size> command or the /OOnn parameter in the BLINKER environment variable.

Example:

```
* Clear all inactive overlays
*
BLIOVLCLR()
```

See Also:

Function : **Get size of overlay pool memory.**

Syntax : **nlValue = BLIOVLOPS()**

Return : **Overlay Pool Operating Size in bytes**

This function returns the operating size of the overlay pool in bytes.

This is set at link time using the BLINKER OVERLAY OPSIZE <size> command, or at run time using the /OOnn setting in the BLINKER environment variable. (see Chapter 9, BLINKER OVERLAY OPSIZE).

The figure returned by this function represents the requested size of the overlay area at the time the program started running. This value will be the most recent of the value set at link time with the BLINKER OVERLAY OPSIZE nnn command and the /OOnnn setting of the BLINKER environment variable at runtime. The actual size of the area allocated may vary depending on the availability of EMS and UMBs and the settings of the appropriate BLINKER options. See the function BLIOVLSIZ() for details of the actual size allocated.

CA-Clipper Summer '87 DYNAMIC method note:

When using the BLINKER OVERLAY DYNAMIC (the default) method of overlay allocation, this size is the maximum amount of memory which Blinker will allocate from CA-Clipper for overlays. Once Blinker has used this much memory for overlays, it will discard unused overlays to free up space for the next overlay, rather than allocating more dynamic memory from CA-Clipper to store the overlay.

During execution the overlay manager may be forced to increase this value if overlays become heavily nested, since all active overlays must remain in memory with the DYNAMIC method. The new value will remain current until the program is ended unless a BLIOVLCLR() is executed. In this case the OPSIZE, and therefore the value returned by the BLIOVLOPS() function, will be reset to the value defined at start-up.

Example:

```
* Get the overlay pool operating size.
*
? "The overlay pool operating size is "
?? BLIOVLOPS(), "bytes"
```

See Also:

Function : **Resume overlay manager operation**

Syntax : **BLIOVLRES()**

Return : **Nothing**

This function resumes overlay manager operations following an overlay manager suspend call. The overlay manager will reinitialise itself, reopen the overlay file(s), and resume processing overlay calls.

See Also:

Function : **Get current size of overlay pool memory.**

Syntax : **nlValue = BLIOVLSIZ()**

Return : **Overlay Pool Current Size**

This function returns the size of the overlay area as it was actually allocated when the program started running. The default size is the most recent of the value set at link time with the BLINKER OVERLAY OPSIZE nnn command and the /OOnnn setting of the BLINKER environment variable at runtime. This size may be modified in one of two ways:

If the BLINKER OVERLAY PAGEFRAME ON command was used at link time, or the /OP1 setting in the BLINKER environment variable is set at runtime, then if an EMS pageframe is available and the default OPSIZE is less than 64kb then the size of the overlay area will be expanded to use as much of the pageframe as possible. This new size will then be the size returned by the BLIOVLSIZ() function.

If the use of the pageframe is not possible or has been disabled, and provided the use of UMBs has not been disabled through the use of the BLINKER OVERLAY UMB OFF command at link time or the /OU0 setting of the BLINKER environment variable at runtime, then Blinker will attempt to allocate a UMB at least as large as the default OPSIZE. If this is successful then the size of the overlay area will be expanded to use as much of the UMB as possible. This new size will then be the size returned by the BLIOVLSIZ() function.

If neither of these methods is possible, or both have been disabled, then the size allocated will be the default OPSIZE.

CA-Clipper Summer '87 DYNAMIC method note:

When using the BLINKER OVERLAY DYNAMIC method of overlay allocation, this value reflects the exact amount of memory being used by Blinker for overlays at the current time. At the start of an application this will be zero, and the value will increase until it reaches the operating size set by BLINKER OVERLAY OPSIZE <size> and returned by BLIOVLOPS().

Once it has reached this size it will stabilize for the rest of the application until a call to BLIOVLCLR(). When a call to BLIOVLCLR() is made, all inactive overlays are discarded and the memory is returned to CA-Clipper, so the current size will decrease to the size of just the active overlays. In the best case of BLIOVLCLR() being called from the root then BLIOVLSIZ() should actually return to zero.

Example:

```
* Get the current overlay pool size.
*
? "The current overlay pool size is"
?? BLIOVLSIZ(), " bytes"
```

See Also:

 # (comment)   A script file comment

Purpose : **A script file comment**

Syntax   : **# ...**

Example : **# To link or not to link, that is the question.**

The remainder of all lines in the script file containing the character # are treated as comments and are ignored by Blinker.

See Also:

Function : **Suspend overlay manager operations**

Syntax : **BLIOVLSUS(niValue)**

Return : **Nothing**

This function shuts down the overlay manager. Depending on the level of shutdown required, the overlay manager will free up some or all of the system resources which it is using. The valid parameters are the numeric values 1 through 3, and the system resources freed for each are as follows:

1 Closing all open overlay files, freeing up the DOS handles.
2 As (1) above, plus freeing of the EMS/XMS used for the overlay cache
3 Freeing ALL resources used by the overlay manager including the overlay pool.

When the overlay manager is resumed using the BLIOVLRES() function, it will attempt to reallocate the same resources it had before. If it is unable to reallocate an overlay area, then the program will terminate with a Blinker error 1216, 'Overlay manager unable to resume'. If the overlay area is reallocated successfully, but there are no resources available for the overlay cache, then execution will continue without caching of overlays.

Note: This routine must be called from the root, as the overlay manager will be inactive on return from this call until a call to BLIOVLRES() is made.

See Also:

BLIPTRDEC() **Huge pointer decrement**

Function : **Huge pointer decrement**

Syntax : **BLIPTRDEC(pointer, nlValue)**

Return : **Decremented pointer**

The BLIPTRDEC() function (decrement huge pointer) is the inverse of the BLIPTRINC() function - for more information on the use of these functions please see the entry for BLIPTRINC() below.

See Also:

BLIPTRINC()   Huge pointer increment

Function : **Huge pointer increment**

Syntax   : **BLIPTRINC(pointer, nlvalue)**

Return   : **Incremented pointer**

Some C/C++ compilers generate code using huge protected mode pointers quite happily (such as Borland C++ in most cases), while others generate references to __AHSHIFT and __AHINCR. These symbols are used to advance huge pointers over segment boundaries, and are provided in the compiler runtime library as constants that work correctly in real mode only. As it is impossible for Blinker to provide single values for these constants that will work in both real mode and protected mode, there are two possible solutions:

1) BLXSHFPT.OBJ or BLXSHFTR.OBJ can be linked as a FILE to resolve these symbols with protected mode or real mode compatible values respectively, but the resulting .EXE will ONLY run correctly IN PROTECTED MODE as the increments are hard coded for protected mode.

2) The code can be updated to use two Blinker functions BLIPTRINC() and BLIPTRDEC() to advance the pointers correctly in all modes, including dual mode.

For example if BLOCK is some complex type then :

```
1  memptr = (BLOCK huge *) halloc ((long) BLKMAX,sizeof (BLOCK));
2  if (memptr)
3     {
4     tmpptr = memptr;
5     for (i = 0; i < BLKMAX; i++)
6        {
7        ...      /* Some operation */
8        tmpptr++;      Borland handles huge pointers ok */
9        }
10    }
```

Line 8 can be replaced with

```
tmpptr = BLIPTRINC (tmpptr, (long) sizeof (BLOCK));
```

See Also:

BLISERNUM() / BLISERNUMBAS() Get burned in serial number

Function : **Get burned in serial number**

Syntax   : **cValue = BLISERNUM()**

Return   : **Serial number character string**

This function is used to examine a serial number and other information of up to 100 characters encrypted into the executable file at link time, for example to hide a serial number and licensee number in the .EXE file as a form of copy protection.

The information is stored as a string of up to 100 characters in length, which is copied directly into the .EXE file and returned in exactly the same format at runtime.

For example, to encrypt a program with a serial number of 00100001, a 5 user license and a licensee name of Blink, Inc. use the following command:

```
BLINKER EXECUTABLE SERIAL 00100001 5 Blink, Inc.
```

At program runtime these values can be examined by assigning the serial number to a string and then manipulating it using a substring function, for example:

```
TMPSERNUM = BLISERNUM()
? "Serial number:", SUBSTR(TMPSERNUM,1,8)
? "No. of Users :", VAL(SUBSTR (TMPSERNUM,10,1))
? "Licensed to :", SUBSTR(TMPSERNUM,12,20)
```

See Also:

Function : **Get CA-Clipper Summer '87 free pool fragmentation as a string**

Syntax : **cValue = BLISTRFRG([cString])**

Return : **Fragmentation as a string**

This function returns a string containing a summary of CA-Clipper Summer '87's dynamic memory to show the degree of fragmentation. Each character position represents up to 512 bytes, and adjacent equal characters represent contiguous blocks larger than 512 bytes.

The optional parameter is a five character string in the form "ffuua" as described below. The first two characters (ff) are used to represent free blocks and the next two characters (uu) to represent used blocks, with the final character (a) used to represent the next allocation point. The returned string contains a map of both the near heap and the far heap (FREE POOL), separated by a ',' character. The layout and meaning of the display is detailed in the discussion of the function BLIDISFRG().

Example:

```
* Get the string showing memory fragmentation
* Use 'f' for all free blocks, 'u' for all used
*
TMPSTRFRG = BLISTRFRG("ffuua")
? "The state of dynamic memory is currently"
? TMPSTRFRG
```

See Also:

// (comment)   A script file comment

Purpose : **A script file comment**

Syntax   : **// ...**

Example : **// To link or not to link, that is the question.**

The remainder of all lines in the script file containing the characters //
are treated as comments and are ignored by Blinker.

See Also:

[Next Entry>>]

BLIVERNUM() **Return the Blinker version number as an integer**

Function : **Return the Blinker version number as an integer**

Syntax : **nuValue := BLIVERNUM()**

Return : **Blinker version used to link the program.**

This function returns an integer containing the version number of Blinker used to link the program. The number represents major version * 100 + minor version, e.g. 300 represents version 3.00.

Example:

```
* Get the Blinker version number
*
? "The Blinker version number is "
?? STR (BLIVERNUM()/100, 2)
```

[^^Up^^]

[<<Previous Entry]

DOS Extender functions

The following functions are for use with the Blinker DOS extender. They will seldom have meaningful effects or return meaningful information except when used with the DOS extender.

In the subsequent discussions and examples, the following terms are used:

Descriptor **A structure containing the size, base and attributes of a protected mode segment.**

Paragraph **The segment portion of a real mode address, eg the paragraph of the colour adapter is B800h.**

Segment **The actual code or data unit as allocated and stored in memory.**

Selector **The value used to identify a segment when running in protected mode, i.e. that is passed to and from the protected mode memory allocation functions.**

All functions assume the following definitions from the include file BLX286.H which can be found in the C subdirectory of the Blinker installation directory:

```
typedef unsigned char BYTE;
typedef unsigned short SEL;
typedef unsigned short USHORT;
typedef unsigned long ULONG;
typedef unsigned short _far *PSEL;
typedef struct
   {
   ULONG segbase;    /* Segment linear base address */
   ULONG seglen;        /* Segment length in bytes */
   USHORT segattrib; /* Segment attribute bytes */
   } DESC;              /* This is not a real descriptor */
#define BLXAPI pascal _far
```

Function : **Allocate an extended memory block of up to 16MB.**

Syntax : **USHORT BLXAPI DosAllocHuge(USHORT segcount, USHORT lastsize, PSEL selector, USHORT segmax, USHORT flags);**

segcount **Count of full 64kb segments to allocate.**

lastsize **Size of the last segment in bytes (0-65535).**

selector **Pointer to an integer to receive the initial selector for the block.**

segmax **Maximum number of 64kb blocks which may later be allocated to the block in a call to DosReallocHuge(). (This can be 0, see below).**

flags **Reserved. Must be zero.**

Return : **The base selector of the huge block.**

DosAllocHuge() allocates a memory block of up to 16MB and returns a selector that points to the base of the block. When the block is larger than 64kb a contiguous block of selectors is allocated and one selector is assigned to each complete or partial segment. The increment used to move from one selector to the next is calculated using DosGetHugeShift().

Segmax specifies the maximum number of 64kb blocks that the block can be resized to. Enough contiguous selectors are allocated by DosAllocHuge() to ensure subsequent reallocations will succeed. Specifying a segmax value of 0 will cause DosAllocHuge() to compute a segmax value equal to the required inital allocation size, so preventing the block from being reallocated beyond (segcount+1) 64kb segments.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main(void)
{
SEL         selector;
USHORT      hshift ;
int         selinc ;
char        *fp;
/* Allocate a 1MB block */
if (DosAllocHuge(16,0,&selector,0,0) != 0)
   {
   printf("DosAllocHuge() failed\n");
   exit(255);
   }
DosGetHugeShift(&hshift);
selinc = 1 << hshift ;
/* fill each 64kb block with NULLS */
tmpsel = selector;
for (cnt = 0, cnt <16; cnt++,tmpsel += selinc);
   {
   fp = MK_FP(tmpsel,0);
   *fp = 0;          /* Fill first character */
   fp++;                /* Advance to next character */
   memset(fp, 0, 0xFFFFh);
   printf("Cleared segment : %04X\n",tmpsel);
   }
/* Free the block */
```

```
    DosFreeSeg(selector);
    }
```

See Also:

Purpose : **Specify the name of a script file**

Syntax   : **@<scriptname>**

Example : **# Process the link script 'Myscript' here**
           @Myscript

Blinker supports nested script files of any size to a depth of 5 levels. The processing of nested script files is similar to a program function call: when Blinker encounters a nested script command, the current script file is suspended, the second script is processed, and then Blinker returns to the first script and resumes processing.

When searching for a script, Blinker will first look in the current directory, and then in the path specified in the LIB environment variable. This will allow frequently used scripts which are rarely modified, (such as scripts for overlaying a particular library), to be stored in a common directory for convenience.

Function : **Allocate a conventional memory block.**

Syntax : **USHORT BLXAPI DosAllocRealSeg(ULONG size, PUSHORT paragraph, PSEL selector);**

size **Size of the block to allocate in bytes (Cannot be greater than 64 Kb).**

paragraph **Pointer to an integer to receive the real mode segment of the memory block.**

selector **Pointer to an integer to receive the protected mode selector of the memory block.**

Return : **The real mode paragraph address and the protected mode selector of the conventional memory block.**

As this function returns both a real mode paragraph address and a protected mode pointer the allocated block can be addressed in either mode, and can therefore be useful in mixed-mode programming.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main(void)
{
SEL         selector;
USHORT      paragraph;
/* Allocate a 16KB block */
if (DosAllocRealSeg((16L << 10), &paragraph, &selector) != 0)
   {
   printf("DosAllocRealSeg() failed\n");
   exit(255);
   }
printf("Real segment: %04X prot selector: %04X\n",
   paragraph,selector);
/* Free the block */
DosFreeRealSeg(selector);
}
```

See Also:

Function : **Allocate an extended memory block of up to 64 Kb.**

Syntax : **USHORT BLXAPI DosAllocSeg(USHORT size, PSEL selector, USHORT flags);**

size **Size of block in bytes (max=64kb).**

selector **Pointer to an integer to receive the selector.**

flags **Reserved (must be 0).**

Return : **Selector for the extended memory block.**

DosAllocSeg() allocates a memory block up to 64kb in size.

Most high level languages provide a memory allocation function equivalent to this function, such as malloc() in C. These functions can be used instead of DosAllocSeg() to manage blocks of memory less than 64kb in size, particularly when minimum source code differences between real and protected mode are required.

Use DosMemAvail() to find the amount of available memory that can be allocated using DosAllocSeg().

Example :

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main(void)
{
SEL         selector;
/* Allocate a 4KB block */
if (DosAllocSeg(4096, &selector, 0) != 0)
   {
   printf("DosAllocSeg() failed\n");
   exit(255);
   }
printf("4K block allocated at : %04X\n",selector);
/* Free the block */
DosFreeSeg(selector);
}
```

See Also:

Function : **Create an executable (code) selector corresponding to a data selector.**

Syntax : **USHORT BLXAPI DosCreateCSAlias(SEL datasel, PSEL codesel);**

datasel **Data selector to be aliased. This selector must belong to a single data segment less than or equal to 64kb.**

codesel **Pointer to an integer to receive the code selector (alias).**

Return : **Executable selector corresponding to data selector. The returned selector has the same linear base address and limit as the original data selector, but has a read / execute code attribute.**

This function allows code to be executed in a data segment.

Note: The original data selector should be locked in memory (using DosLockSeg()) prior to creating the alias selector. This ensures that the original selector will not be swapped out of memory, which could later cause the linear base address of the segment to change when the segment is reloaded from disk. This is necessary because alias selectors do not track changes to the original selector. The original data selector may be unlocked by calling DosUnlockSeg() after the alias selector has been freed using DosFreeSelector().

Example:

```
#include <stdio.h>
#include <blx286.h>
#define TOSCREEN  2
#define TOPRINT   5

unsigned char codebyte[] =
   {
   0xb8, 0x00, 0x00,       /* mov AX, 0 */
   0xba, 0x00, 0x00, /* mov DX, 0 */
   0xcd, 0x21,  /* int 21 */
   0xcb   /* retf */
   };
void (far *execdata) (void);
void output(char *string, unsigned char device)
   {
   while (*string)
      {
      codebyte[2] = device;      /* screen or printer */
      codebyte[4] = *string;  /* char to output */
      (*execdata)();
      string++;
      }
   }
void main(void)
{
/* This is a contrived example */
void far *fp;
SEL         codesel ;
/* Lock the segment first */
fp = (void far *)(codebyte);
DosLockSeg(SELECTOROF(fp));
if (DosCreateCSAlias(SELECTOROF(fp),&codesel) == 0)
   {
   execdata = MK_FP(codesel, OFFSETOF(fp));
   output("This is a contrived example\n",TOSCREEN);
   output("This is a contrived example\n",TOPRINT);
```

```
        DosFreeSelector(codesel);
        }
    }
```

See Also:

DosCreateDSAlias() Create a data selector corresponding to an executable (code) selector.

Function : **Create a data selector corresponding to an executable (code) selector.**

Syntax : **USHORT BLXAPI DosCreateDSAlias(SEL codesel, PSEL datasel);**

codesel **Code selector to be aliased.**

datasel **Pointer to an integer to receive the data selector (alias).**

Return : **Data selector corresponding to an executable code selector. The returned selector has the same linear base address and limit as the original code selector, but has a read / write code attribute.**

Note: The original code selector should be locked in memory (using DosLockSeg()) prior to creating the alias selector. This ensures that the original selector will not be swapped out of memory, which could later cause the linear base address of the segment to change when the segment is subsequently reloaded from disk. This is necessary because alias selectors do not track changes to the original selector. The original code selector may be unlocked by calling DosUnlockSeg() after the alias selector has been freed using DosFreeSelector().

This function allows code segments to be written to.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main(void)
{
SEL      datasel
/* Lock the segment first */
DosLockSeg(SELECTOROF(test));
if (DosCreateDSAlias (SELECTOROF (test), &datasel) != 0)
   {
   printf ("DosCreateDSAlias() failed\n");
   exit (255);
   }
printf ("Code selector for test() is %04X\n",
   SELECTOROF (test));
printf ("Data selector is %04X\n",datasel);
DosFreeSelector(datasel);
}
```

See Also:

DosFreeRealSeg() **Free a selector previously allocated with DosAllocRealSeg()**

Function : **Free a selector previously allocated with DosAllocRealSeg()**

Syntax : **USHORT BLXAPI DosFreeRealSeg(SEL selector);**

selector **Protected mode selector of a conventional memory block allocated through DosAllocRealSeg().**

Return : **None**

This function is used to free conventional memory blocks allocated through DosAllocRealSeg(), and should not be used to free any other type of selector.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main(void)
{
SEL         selector;
USHORT      paragraph;
/* Allocate a 4 Kb block */
if (DosAllocRealSeg(4096, &paragraph, &selector) != 0)
   {
   printf("DosAllocRealSeg() failed\n");
   exit(255);
   }
printf("4 Kb conventional block allocated\n");
printf("Selector for block : %04X\n",selector);
printf("Segment of block : %04X\n",paragraph);
if (DosFreeRealSeg(selector) != 0)
   {
   printf("DosFreeRealSeg() failed - selector = %04X\n",
      selector);
   exit(254);
   }
}
```

See Also:

Function : **Free a memory block allocated with DosAllocSeg() or DosAllocHuge().**

Syntax : **USHORT BLXAPI DosFreeSeg(SEL selector);**

selector **Protected mode selector of a memory block allocated through DosAllocSeg() or DosAllocHuge().**

Return : **None**

This function is used to free memory blocks allocated through DosAllocSeg() or DosAllocHuge() and should not be used to free any other type of selector.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
SEL         selector;
/* Allocate a 4KB block */
if (DosAllocSeg(4096, &selector, 0) != 0)
   {
   printf("DosAllocSeg() failed\n");
   exit(255);
   }
printf("4 Kb block allocated at : %04X\n",selector);
if (DosFreeSeg(selector) != 0)
   {
   printf("DosFreeSeg() failed - selector = %04X\n",
      selector);
   exit(254);
   }
}
```

See Also:

Function : **Free a selector allocated with DosMapRealSeg(), DosCreateCSAlias() or DosCreateDSAlias()**

Syntax : **USHORT BLXAPI DosFreeSelector(SEL selector);**

selector **Selector allocated through DosMapRealSeg(), DosCreateCSAlias() or DosCreateDSAlias().**

Return : **None**

This function is used to free alias selectors (created with DosCreateDSAlias() or DosCreateCSAlias()) or selectors created using DosMapRealSeg(), and should not be used to free any other type of selector.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>

#define VID_BUF 0xB800
#define VID_SIZE 0x8000l

unsigned char far *VideoBuffer;
void main (void)
{
SEL         selector;
ULONG       screen_address;
/* Get a selector that corresponds to the screen buffer */
if (DosMapRealSeg(VID_BUF,VID_SIZE,&selector) != 0)
   {
   printf("Unable to map screen segment\n");
   exit(255);
   }
VideoBuffer = MK_FP(selector,0);
*VideoBuffer[0] = "X";        /* Write to screen */
if (DosFreeSelector(selector)==0)
   printf("Video buffer selector now invalid\n");
}
```

See Also:

Function : **Get a selector for the 1kb BIOS data area.**

Syntax : **USHORT BLXAPI DosGetBIOSSeg(PSEL biossel);**

biossel **Pointer to an integer to receive the protected mode selector to the BIOS data area.**

Return : **Selector for the BIOS data area corresponding to real mode segment 0x0040.**

Far pointers to the BIOS data area should be created using the selector returned by this function. The selector returned by this function should NEVER be freed.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
SEL      biossel;

/* Get selector for BIOS data area */
if (DosGetBIOSSeg(&biossel) != 0)
   {
   printf("DosGetBIOSSeg() failed\n");
   exit(255);
   }
printf("BIOS selector is : %04X\n",biossel);
}
```

DosGetExceptionHandler()   Get the protected mode address of an exception handler

---

Function : **Get the protected mode address of an exception handler**

Syntax   : **USHORT BLXAPI DosGetExceptionHandler(USHORT exception, PEXCHAN far *handler);**

exception **Exception number (0-13).**

handler   **Pointer to a function pointer to receive the address of the exception handler.**

Return   : **Address of the exception handler.**

DosGetExceptionHandler() returns the address of the current handler for the specified exception number.

Example:

```
#include <stdio.h>
#include <blx286.h>
void main(void)
{
EXCHAN   handler;
/* Get current exception handler */
if (DosGetExceptionHandler(0x0D, &handler) == 0)
   printf("Exception 13 handler : %Fp\n",handler);
}
```

See Also:

Function : **Get the current DOS extender host mode (DPMI, VCPI, XMS).**

Syntax : **USHORT BLXAPI DosGetHostMode(BYTE *hostmode);**

hostmode **Pointer to a byte to receive the host mode.**

Return : **Host mode indicator byte.**

This function is used to determine under which host mode the DOS extended program is running.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
BYTE        hostmode ;
if(DosGetHostMode(&hostmode) == 0 )
   {
   printf("The host mode is ");
   switch(hostmode)
      {
      case BliHostDPMI :
         printf("DPMI");
         break;
      case BliHostVCPI :
         printf("VCPI");
         break;
      case BliHostXMS :
         printf("XMS");
         break;
      default :
         printf("Unknown!!");
      }
   }
}
```

Function : **Get the left shift count to apply to 1 to get the huge selector increment value.**

Syntax : **USHORT BLXAPI DosGetHugeShift(PUSHORT shiftcount);**

shiftcount **Pointer to an integer to receive the huge shift value.**

Return : **Left-shift count integer.**

The huge increment value is obtained by applying the left-shift count returned by this function to1. For example, a left-shift count of 3 gives a huge increment of 8 (1 << 3 == 8). This increment is added to the base selector returned by DosAllocHuge() to give the next consecutive selector for a huge memory block.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
USHORT      shiftcount;
int         increment;
/* Get the huge shift count */
if (DosGetHugeShift(&shiftcount) != 0);
   {
   printf("DosGetHugeShift() failed\n");
   exit(255);
   }
increment = 1 << shiftcount;
printf("Huge shift : %04X\n",shiftcount);
printf("Huge increment : %04X\n",increment);
}
```

Purpose : **Specify to the linker it is running in batch mode**

Syntax : **BATCH**

Default : **Enabled**

The BATCH command is used to specify to the linker that it is being run in batch mode. If a particular file or library cannot be found in the appropriate path then linking terminates with an error, rather than allowing an interactive alternative. Blinker ignores the BATCH command as it is the default mode of operation for Blinker.

DosGetMachineMode()   Get the current machine mode (real, 286 protected).

Function : **Get the current machine mode (real, 286 protected).**

Syntax   : **USHORT BLXAPI DosGetMachineMode(BYTE *mmode);**

mmode   **Pointer to a byte to receive the machine mode.**

Return   : **Machine mode indicator byte**

DosGetMachineMode() returns the current machine mode (either real or protected). Since Blinker has the capability of creating dual mode programs that may run in protected or real mode depending on the host environment, DosGetMachineMode() may be used to determine the current mode and take appropriate action.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
unsigned char far *VideoBuffer;
void main(void)
{
BYTE        machinemode;
SEL         selector;
if (DosGetMachineMode(&machinemode) == 0)
   switch (machinemode)
      {
      case MODE_REAL:
         printf ("\nReal mode - absolute pointers ok\n");
         VideoBuffer = MK_FP(0xb800,0);
         break;
      case MODE_PROTECTED:
         printf ("\n286 Protected mode - plenty memory\n");
         if (DosMapRealSeg (0xb800, 0x8000l, &selector))
            {
            printf ("\nTrouble\n");
            exit (1);
            }
         VideoBuffer = MK_FP(selector,0);
         break;
      default:
         printf ("\nOther mode - interesting\n");
         exit(1);
      }
}
```

Then:

```
/* Display a '*' in top left corner of screen*/
*VideoBuffer[0] = '*';
```

Function : **Get a protected mode pointer to a protected mode interrupt handler**

Syntax : **USHORT BLXAPI DosGetProtVec(USHORT intno, PINTHAN far *handler)**

intno **Interrupt number.**

handler **Pointer to a function pointer to receive the address of the interrupt handler.**

Return : **Address of the interrupt handler.**

DosGetProtVec() returns the protected mode address of the protected mode interrupt handler for the specified interrupt number.

Example:

```
#include <stdio.h>
#include <blx286.h>
void main(void)
{
INTHAN      handler;
if (DosGetProtVec(0x21, &handler) == 0)
   {
   printf("Protected mode Int 21 handler : %Fp\n", handler);
   }
}
```

See Also:

DosGetRealVec() **Get a real mode pointer to a real mode interrupt handler**

Function : **Get a real mode pointer to a real mode interrupt handler**

Syntax : **USHORT BLXAPI DosGetRealVec(USHORT intno, PINTHAN far *handler);**

intno **Interrupt number.**

handler **Pointer to a function pointer to receive the real mode address of the interrupt handler.**

Return : **Address of the interrupt handler.**

DosGetRealVec() returns the real mode address of the real mode interrupt handler for the specified interrupt number.

Example:

```
#include <stdio.h>
#include <blx286.h>
void main(void)
{
INTHAN handler;
if (DosGetRealVec(0x21, &handler) == 0)
   {
   printf("Real mode Int 21 handler : %Fp\n",handler);
   }
}
```

See Also:

Function : **Get the base address of a segment selector.**

Syntax : **USHORT BLXAPI DosGetSegBase(SEL selector, PULONG baseaddr);**

selector **Unsigned value representing a valid selector number.**

baseaddr **Pointer to a long integer to receive the returned selector base address.**

This function returns the linear base address of the selector parameter.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>

#define VID_BUF 0xB800
#define VID_SIZE 0x8000l

void main (void)
{
SEL         video_buffer;
ULONG       screen_address ;
/* get selector that corresponds to the screen buffer */
if (DosMapRealSeg(VID_BUF,VID_SIZE,&video_buffer) != 0)
   {
   printf("Unable to map screen segment\n");
   exit(255);
}
printf("Selector for video buffer %04X\n",video_buffer);
if (DosGetSegBase(video_buffer, &screen_address) == 0)
   printf("Screen base address : %08X\n",screen_address);
}
```

Function : **Get the descriptor structure of a segment selector.**

Syntax : **USHORT BLXAPI DosGetSegDesc(SEL selector, PDESC segdesc);**

selector **A valid selector number whose descriptor is required.**

segdesc **Pointer to a DESC structure (see below) to receive the descriptor.**

Return : **Simplified segment descriptor containing the segment's base address, length and attributes as defined in the BLX286.H header file.**

This function is used to obtain all the information about a particular selector.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
DESC        segdesc;
if (DosGetSegDesc(SELECTOROF(main),&segdesc) == 0)
   {
   printf ("Current code segment base address is %08Xh\n",
      segdesc.segbase);
   printf ("Current code segment length is %08Xh\n",
      segdesc.seglen);
   printf ("Current code segment attributes are %04Xh\n",
      segdesc.segattrib);
   }
}
```

DosGetSegLimit() **Get the size limit of a segment selector.**

Function : **Get the size limit of a segment selector.**

Syntax : **USHORT BLXAPI DosGetSegLimit(SEL selector, PULONG seglimit);**

selector **A valid selector number.**

seglimit **Pointer to a long integer to receive the size limit of the selector.**

Return : **The segment limit of the specified selector.**

The segment limit is the highest offset within a segment that can be addressed.

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>

#define VID_BUF 0xB800
#define VID_SIZE 0x8000l

void main (void)
{
SEL         video_buffer;
ULONG       seglimit;
/* Get selector that corresponds to the screen buffer */
if (DosMapRealSeg(VID_BUF,VID_SIZE,&video_buffer) != 0)
   {
   printf("Unable to map screen segment\n");
   exit(255);
   }
printf("Selector for video buffer %04X\n",video_buffer);
if (DosGetSegLimit(video_buffer, &seglimit) == 0)
   /* Value returned should be 0x7fff in this example */
   printf("Screen segment limit : %08X\n",seglimit);
}
```

Function : **Determine whether the program is running under the Blinker DOS extender.**

Syntax : **USHORT BLXAPI DosIsBlinkX(void);**

Return : **A TRUE or FALSE value indicating whether the program is running under the Blinker DOS extender.**

Example:

```
#include <stdio.h>
#include <blx286.h>
void main (void)
{
if (DosIsBlinkX())
   printf("Running under the Blinker extender\n");
}
```

DosLockSeg() **Lock a segment in memory to prevent it being swapped out by the Blinker DOS e**

Function : **Lock a segment in memory to prevent it being swapped out by the Blinker DOS extender's VM system.**

Syntax : **USHORT BLXAPI DosIsBlinkX(SEL selector);**

selector **The selector to be locked.**

Return : **None**

DosLockSeg() locks the memory associated with the selector parameter in memory, preventing it from being swapped out to disk. This function should be used to lock hardware interrupt handlers into memory, as it may not be possible to call DOS to reload the segment at the time the interrupt occurs. Locking a selector causes the segment to be loaded into memory.

DosLockSeg() should also be used to lock selectors into memory before they are aliased using DosCreateDSAlias() or DosCreateCSAlias(). Segments that are to be aliased using these functions must be locked to prevent them from being swapped out and subsequently reloaded at a different linear base address, which would cause the alias selector to no longer point to the same memory region as the original alias.

Example:

```
#include <stdio.h>
#include <blx286.h>

void comminstall(void);
void commhandler(void);

void main(void)
{
if (DosLockSeg(SELECTOROF(commhandler))==0)
   {
   printf("Serial interrupt handler locked in memory\n");
   comminstall(); /* Hook serial comms interrupt */
   ...
   }
}
```

See Also:

Purpose : **Specify the start of a dynamic overlay area**

Syntax : **BEGINAREA**

The BEGINAREA command is used to specify to the linker the start of an overlay area in the output file. Each BEGINAREA should have a corresponding ENDAREA to indicate the end of the overlay area.

With static overlays BEGINAREAs, and therefore overlay areas, may be nested within one another so long as there is a corresponding ENDAREA for each BEGINAREA. This allows the developer to create a structure of overlays within an area to use the minimum amount of memory for the maximum number of overlay routines.

With Blinker dynamic overlays, as opposed to static overlays, the mutual dependency of each overlay procedure or file is unimportant. Also, the multiple nested overlay areas associated with static overlays are no longer necessary, although multiple overlay areas may be helpful in ordering the included libraries. The BEGINAREA and ENDAREA commands are simply treated as markers to the beginning and end of code which is to be overlaid.

For most compilers, the .OBJ files listed between a BEGINAREA and ENDAREA are managed at runtime on a segment basis, i.e. the whole CODE segment for each .OBJ is loaded into memory whenever any function or procedure within that .OBJ file is called.

For more details on how functions and procedures may be kept in separate segments to reduce the overlay unit size, please refer to the appropriate compiler manual for the language being used.

Note for CA-Clipper:

All CA-Clipper Summer '87 .OBJ files listed between a BEGINAREA and ENDAREA are managed at runtime on a procedure or function basis, no matter how many procedures or functions are present in the .OBJ file. Therefore excessively large procedures and functions should be avoided. The CA-Clipper 5.x dynamic paging system automatically breaks procedures down into 1kb pages for improved execution speed, so this is not a concern.

See Also:

Function : **Convert a real mode paragraph value into a protected mode selector.**

Syntax : **USHORT BLXAPI DosMapRealSeg(USHORT paragraph, ULONG size, PSEL selector);**

paragraph **A real mode segment address, e.g. to map address B800:0000 for the colour screen use 0xB800.**

size **The size of the memory region to be mapped in bytes, which may be larger than 64 Kb.**

selector **Pointer to an integer to receive the selector.**

Return : **A selector which has the same base address as the real mode segment.**

DosMapRealSeg() is used to map real mode segments onto protected mode selectors in order to make real mode addresses accessible from protected mode. (For example the real mode video buffer at segment 0xB800).

If you need to access the low-memory BIOS data area at real mode segment 0x0040 in protected mode, use DosGetBIOSSeg() to obtain a selector in preference to this function.

DosMapRealSeg() does NOT allocate memory, if you need to allocate real mode memory use DosAllocRealSeg().

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>

#define VID_BUF 0xB800
#define VID_SIZE 0x8000l

void main (void)
{
SEL         video_buffer;
/* Get a selector that corresponds to the screen buffer */
if (DosMapRealSeg(VID_BUF,VID_SIZE,&video_buffer) != 0)
   {
   printf("Unable to map screen segment\n");
   exit(255);
   }
printf("Selector for video buffer %04X\n",video_buffer);
}
```

Function : **Get total available extended memory.**

Syntax : **USHORT BLXAPI DosMemAvail(PULONG bytesavail);**

bytesavail **Pointer to a long integer to receive the number of bytes of available extended memory.**

Return : **The total available extended memory in bytes.**

DosMemAvail() returns the amount of extended memory available which may be allocated with DosAllocSeg() and DosAllocHuge().

Example:

```
#include <stdio.h>
#include <blx286.h>
void main (void)
{
ULONG       ext_mem_avail;
if (DosMemAvail(&ext_mem_avail) != 0)
   {
   printf("Unable to determine free extended memory\n");
   exit(255);
   }
printf("Free extended memory is %08X bytes\n",ext_mem_avail);
}
```

See Also:

DosRealAvail() **Get total available real mode memory.**

Function : **Get total available real mode memory.**

Syntax : **USHORT BLXAPI DosRealAvail(PULONG bytesavail);**

bytesavail **Pointer to a long integer to receive the number of bytes of available real mode memory.**

Return : **The total available real mode memory in bytes.**

DosRealAvail() returns the amount of real mode memory available which may be allocated with DosAllocRealSeg().

Example:

```
#include <stdio.h>
#include <blx286.h>
void main(void)
{
ULONG       vmfree, xmfree, rmfree ;
if (DosVMAvail(&vmfree) == 0)
   printf("There are %08X bytes free virtual memory\n", vmfree);
if (DosMemAvail(&xmfree) == 0)
   printf("There are %08X bytes free extended memory\n",    xmfree);
if (DosRealAvail(&rmfree) == 0)
   printf("There are %08X bytes free real memory\n", rmfree);
}
```

See Also:

Function : **Make a far call to a real mode address**

Syntax   : **USHORT _far _cdecl DosRealFarCall(REALPTR realfn, PREGS regs, REALPTR reserved, short wordcount, ...);**

realfn   **Real mode address to call.**

regs   **Pointer to REGS16 structure containing the register values to be passed to and returned from real mode.**

reserved   **Reserved. Must be zero.**

wordcount **Count of word arguments to be passed to realfn on the stack. Zero indicates none and a negative count causes them to be passed using the Pascal calling convention.**

...   **Arguments to realfn if wordcount <> 0.**

Return   : **Register values in regs structure.**

DosRealFarCall() allows real mode code to be called from protected mode via a far call. The real mode code must return via a far return instruction. Arguments may be passed on the stack or in registers via the REGS16 structure. Return values are returned in the REGS16 structure.

Example:

```
#include <stdio.h>
#include <blx286.h>
void main(void)
{
REGS16          regs;
BYTE            far *xmsdriver ;
printf("This program attempts to get the XMS driver ");
printf("version number\n");
printf("using DosRealIntr() and DosRealFarCall()\n\n");
/* Important to clear structure to zero */
memset(&regs, 0, sizeof(REGS16)) ;
regs.ax = 0x4300 ;
if (DosRealIntr(0x2f, &regs, 0L, 0) == 0)
/* XMS installation check */
   if ((regs.ax & 0xff)==0x80)
      {
      printf("XMS driver installed\n");
      /* Get driver entry point */
      regs.ax = 0x4310;
      if (DosRealIntr(0x2f, &regs, 0L, 0) == 0)
         {
         xmsdriver = MK_FP(regs.es, regs.bx);
         /* real mode entry point */
         printf("XMS real mode entry point is : %Fp\n",
            xmsdriver);
         /* Clear register structure to zero */
         memset(&regs, 0, sizeof(REGS16));
         /* Get version function */
         regs.ax = 0;
         if (DosRealFarCall((REALPTR)xmsdriver,
            &regs, 0L, 0) == 0)
            if(regs.ax != 0)
               printf("XMS driver version : %04X\n", regs.ax);
            else
               printf("XMS get version call failed\n");
         else
            printf("DosRealFarCall() failed\n");
```

```
            }
         else
            printf("DosRealIntr() call failed\n");
         }
      else
         printf("XMS driver not installed\n");
   else
      printf("DosRealIntr() call failed\n");
   }
```

See Also:

Function : **Generate a real mode interrupt from protected mode.**

Syntax : **USHORT _far _cdecl DosRealIntr(USHORT intno, PREGS regs ULONG reserved, short wordcount, ...);**

intno **Real mode interrupt number.**

regs **Pointer to REGS16 structure containing the register values to be passed to and returned from real mode.**

reserved **Reserved. Must be zero.**

wordcount **Count of word arguments to be passed to the interrupt handler on the stack. Zero indicates none and a negative count causes them to be passed using the Pascal calling convention.**

... **Arguments to interrupt handler if wordcount <> 0.**

Return : **Register values in regs structure.**

DosRealIntr() is used to issue a real mode interrupt from protected mode. The Blinker extender provides automatic transparent DOS and BIOS interrupt support for most of the common interrupts, so it is only necessary to use this function for non-standard interrupts.Examples of these include Network APIs, undocumented DOS functions and real mode TSRs.

Example:

```
#include <stdio.h>
#include <blx286.h>
void main (void) /* Note: This example could also be */
               /* done directly using INT 10h */
{
REGS16         regs;
regs.ax = 0x0f00;
if (DosRealIntr (0x10h, &regs, 0L, 0) == 0)
   {
   printf ("Current video mode : %02Xh\n", regs.ax && 0xff);
   printf ("Current video cols : %02Xh\n", regs.ax >> 8);
   }
else
   printf ("Get video mode call failed\n");
}
```

See Also:

Function : **Resize an extended memory block of up to 16 MB.**

Syntax : **USHORT BLXAPI DosReallocHuge(USHORT segcount,**
USHORT lastsize, SEL selector);

segcount **New count of full 64kb segments to allocate.**

lastsize **Size of last segment in bytes.**

selector **Selector originally returned by DosAllocHuge() when the memory block was initially allocated.**

Return : **None**

DosReallocHuge() changes the size of a memory block originally allocated with DosAllocHuge().

Huge blocks can never be resized to a size larger than (segmax+1) 64 Kb blocks as specified in the original call to DosAllocHuge().

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
SEL         selector;
/* Allocate a 1MB block (16*64kb) */
/* allowing for expansion to 2MB (32*64kb) */
if (DosAllocHuge(16,0,&selector,32,0) != 0)
   {
   printf("DosAllocHuge() failed\n");
   exit(255);
   }
/* Attempt to reallocate block to 1.5MB (24*64kb) */
if (DosReallocHuge(24,0,selector) != 0)
   {
   printf("DosReallocHuge() failed\n");
   exit(254);
   }
printf("Block successfully expanded to 1.5MB\n");
DosFreeSeg(selector);
}
```

Function : **Resize an extended memory block of up to 64kb.**

Syntax : **USHORT BLXAPI DosReallocSeg(USHORT size, SEL selector);**

size **New size of segment in bytes (0 = 64 Kb).**

selector **Selector originally returned by DosAllocSeg() when the memory block was allocated.**

Return : **None**

DosReallocSeg() changes the size of a memory block originally allocated with DosAllocSeg(). The block may not be reallocated to a size larger than 64kb (use DosAllocHuge()/DosReallocHuge() to allocate/reallocate huge memory blocks).

Example:

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <blx286.h>
void main (void)
{
SEL         selector;
/* Allocate a 4kb block */
if (DosAllocSeg(4096, &selector, 0) != 0)
   {
   printf("DosAllocSeg() failed\n");
   exit(255);
   }
printf("4K block allocated at : %04X\n",selector);
/* Expand block to 8kb */
if (DosReallocSeg(8192, &selector) != 0)
   {
   printf("DosReallocSeg() failed\n");
   exit(254);
   }
printf("Block successfully expanded to 8kb\n");
DosFreeSeg(selector);
}
```

DosSetExceptionHandler()   Install a protected mode exception handler

---

Function : **Install a protected mode exception handler**

Syntax   : **USHORT BLXAPI DosSetExceptionHandler(USHORT excno, PEHANDLER newhandler, PEHANDLER _far *prevhandler);**

excno   **Exception number (0-13).**

newhandler **Pointer to new handler.**

prevhandler**Pointer to a function pointer to receive the address of the previous exception handler.**

Return   : **Address of the previous exception handler.**

DosSetExceptionHandler() installs a protected mode handler for processor exceptions.

The exceptions it is possible to handle are:

```
0x00  Divide by zero
0x01  Debug
0x03  Breakpoint
0x04  Overflow
0x05  Bounds check
0x06  Invalid opcode
0x07  287/387 not available
0x08  Double fault
0x0A  Invalid task switch segment
0x0B  Segment not present
0x0C  Stack fault
0x0D  General protection fault
```

Example :

```
#include <stdio.h>
#include <stdlib.h>
#include <blx286.h>
PEXCHAN prevhandler;
void printstr (char *s)    /* Cannot use C screen i/o */
   {
   char c;
   while (c = *(s++))         /* Until end of string */
      {
      _asm mov ah,0eh
      _asm mov al,c
      _asm int 10h            /* Display teletype */
      }
   }
void _interrupt far gphandler(EXCEP_FRAME ef)
   {
   char        msg [80];
   printstr ("Replacement GPF handler\n\r");
   printstr ("----------------------\n\r");
   sprintf (msg,"Error code = %04x\n\r",ef.error_code);
   printstr (msg);
   sprintf (msg," CS:IP = %04x:%04x\n\r", ef.ret_cs,
      ef.ret_ip);
   printstr (msg);
   DosSetExceptionHandler (0x0d,prevhandler,NULL);
   exit (255);
}
void main (void)
{
PEXCHAN        newhandler;
```

```
    newhandler = gphandler;
    if (DosSetExceptionHandler (0x0d, newhandler,&prevhandler)
       == 0)
       {
       *((USHORT far *) 0x22222222L) = 1;     /* Cause exception */
       printf("Somehow no exception occurred\n");
       DosSetExceptionHandler (0x0d,prevhandler,NULL);
       }
    }
```

See Also:

Function : **Install a protected mode interrupt handler**

Syntax : **USHORT BLXAPI DosSetProtVec(USHORT intno, PIHANDLER newhandler, PIHANDLER _far * prevhandler);**

intno **Interrupt number.**

newhandler **Pointer to new handler.**

prevhandler **Pointer to a function pointer to receive the address of the previous interrupt handler.**

Return : **Address of the previous interrupt handler.**

DosSetProtVec() installs a protected mode interrupt handler for the specified protected mode interrupt. DosSetProtVec() also returns a pointer to the previous interrupt handler.

Example:

```
#include <stdio.h>
#include <blx286.h>
USHORT ticks;
PINTHAN prevhandler;
void interrupt far timer1c(REGS16 r);
   {
   ticks++;
   _chain_intr(&prevhandler);
   }
void main(void)
{
PINTHAN     newhandler;
int         cnt0,cnt1;
ticks = 0;
newhandler = timer1c;
if (DosSetProtVec(0x1c, newhandler, &prevhandler) == 0)
   {
   printf("User timer hooked successfully\n");
   printf("Waiting a few seconds\n");
   for (cnt0 = 0; cnt0 < 500; cnt0++)
      for (cnt1 = 0; cnt1 < 32000; cnt1++);
   if (DosSetProtVec(0x1c, prevhandler, NULL) == 0)
      printf("Timer handler deinstalled\n");
   else
      printf("Unable to deinstall timer handler\n");
   printf("Number of protected mode ticks : %04X\n", ticks);
   }
else
   printf("Unable to hook timer tick.\n");
}
```

See Also:

DosSetRealVec() **Install a real mode interrupt handler**

Function : **Install a real mode interrupt handler**

Syntax : **USHORT BLXAPI DosSetRealVec(USHORT intno,**
REALPTR newhandler, REALPTR _far *prevhandler);

intno **Interrupt number.**

newhandler **Pointer to new handler.**

prevhandler **Pointer to a function pointer to receive the address of the previous interrupt handler.**

Return : **Address of the previous interrupt handler.**

DosSetRealVec() sets the pointer for the specified real mode interrupt in the real mode interrupt vector table. DosSetRealVec() also returns a pointer to the previous interrupt handler.

An example of how to use this function can be found in the files INTMAIN.C, INTHAND.ASM and INTMAIN.LNK in the C subdirectory of where Blinker is installed.

See Also:

DosSetSegAttrib() **Set segment attributes for a selector**

Function : **Set segment attributes for a selector**

Syntax : **USHORT BLXAPI DosSetSegAttrib(SEL selector, USHORT attr);**

selector **Selector of segment to modify.**

attr **New attributes for segment.**

DosSetSegAttrib() can be used to alter the type of a segment from code to data or from data to code, and also to modify the read/write/execute attributes of the segment. The allowable attributes are declared in BLX286.H.

Unlike DosCreateCSAlias() and DosCreateDSAlias(), DosSetSegAttrib() does NOT create a new descriptor - instead the existing descriptor is modified directly.

Example:

```
#include <stdio.h>
#include <blx286.h>
unsigned char far array[] = { 1, 2, 3, 4};
void main(void)
{
/* Set array segment to read only */
if (DosSetSegAttr(SELECTOROF(array[0]), DATA16_NOWRITE) == 0)
   printf("Segment containing array is now read only\n");
/* Put it back */
if (DosSetSegAttr(SELECTOROF(array[0]), DATA16) == 0)
   printf("Segment is now read/write\n");
}
```

See Also:

Function : **Unlock a segment previously locked using DosLockSeg()**

Syntax : **USHORT BLXAPI DosUnlockSeg(SEL selector);**

selector **A selector previously locked using DosLockSeg().**

Return : **None**

DosUnLockSeg() unlocks the memory associated with the selector parameter allowing it to be swapped out to disk. This function should be used to unlock hardware interrupt handlers when they are no longer active, and to unlock any selector whose alias has been freed.

Example:

```
#include <stdio.h>
#include <blx286.h>

void comminstall(void);
void commhandler(void);
void communinstall(void);

void main(void)
{
if (DosLockSeg(SELECTOROF(commhandler)) == 0)
   {
   printf("Serial interrupt handler locked in memory\n");
   comminstall(); /* Hook serial comms interrupt */
   ...
   communinstall(); /* Unhook comms interrupt */
   if (DosUnlockSeg(SELECTOROF(commhandler)) == 0)
      printf("Comms handler may now be swapped to disk\n");
   }
}
```

See Also:

Function : **Get the read/write/code/data flags for a selector**

Syntax   : **USHORT BLXAPI DosVerifyAccess(SEL selector,**
PUSHORT flags);

selector   **Selector (or any unsigned value).**

flags   **Pointer to an integer to receive the attribute flags.**

Return   : **The segment flags.**

DosVerifyAccess() is used to determine if the specified selector value is a valid selector, and if so its type (code or data) and access rights (read or write).

Example:

```
#include <stdio.h>
#include <blx286.h>
void DumpFlags(SEL sel)
   {
   USHORT   flags;
   if (DosVerifyAccess(sel, &flags) == 0)
      if (flags & SEL_VALID) /* If a valid selector */
         {
         if (flags & SEL_WRITABLE) printf("WRITABLE ");
         if (flags & SEL_READABLE) printf("READABLE ");
         printf("%s\n", (flags & SEL_CODE) ? "CODE" : "DATA");
         }
      else
         printf("NOT A SELECTOR\n");
   }
void main(void)
{
SEL         selector;
if (DosLockSeg(SELECTOROF(main)) == 0)
   if (DosCreateDSAlias(SELECTOROF(main),&selector) == 0)
      {
      printf ("Code selector for main() is %04X\n",
         SELECTOROF (main));
      printf ("Data selector is  %04X\n", selector);
      printf ("Code selector flags : ");
      DumpFlags(SELECTOROF(main));
      printf ("Data selector flags : ");
      DumpFlags(selector);
      }
}
```

See Also:

[Next Entry>>]

```
DosVMAvail()                    Get the amount of available virtual memory
------------------------------------------------------------------------------
```

Function : **Get the amount of available virtual memory**

Syntax : **USHORT DosVMAvail(PULONG bytes);**

bytes **Pointer to a long integer to receive number of bytes of available virtual memory.**

Return : **The total available virtual memory in bytes.**

DosVMAvail() returns the approximate number of bytes of virtual memory available to the protected mode program.

Example:

```
#include <stdio.h>
#include <blx286.h>
void main(void)
{
ULONG       vmfree, xmfree, rmfree;
if (DosVMAvail(&vmfree) == 0)
   printf("There are %08X bytes free virtual memory\n",
      vmfree);
if (DosMemAvail(&xmfree) == 0)
   printf("There are %08X bytes free extended memory\n",
      xmfree);
if (DosRealAvail(&rmfree) == 0)
   printf("There are %08X bytes free real memory\n",
      rmfree);
}
```

See Also:

[^^Up^^]

[<<Previous Entry]

Swap / run package functions

The swap / run package functions use the C calling conventions as described above in the Blinker Programming Functions section.

The swap / run functions are provided with CA-Clipper, BASIC and C interfaces which are defined in the appropriate language include files BLINKER.BI, BLINKER.H, BLINKER.CH, BLINKER.FI, BLINKER.HDR, BLINKER.DEF and BLINKER.INC, which can be found in the BAS, C, CLP, FOR, FRC, MOD and PAS subdirectories respectively of the Blinker installation directory:

Function : **Increase size of child environment block.**

Syntax : **niValue = SWPADDENV(NumberOfBytes)**

Return : **The previous setting**

By default, when the DOS EXEC function starts a process (SWPRUNCMD()

uses this DOS function), the environment that is created for the child is truncated so that there is very little, if any, free space.

This function can be used to increase the size of the child program's environment by the number of bytes passed as a parameter. This is necessary if the program (or batch file) that will be run as a child process needs to add or change environment variables. The maximum size of an environment block is 32 KB.

Note that this function increases the size of the child environment at the time the child process is executed, it does not modify the parent environment in any way.

The child environment is destroyed when the child process terminates.

Example:

```
// Ensure 128 bytes free in environment
OldSize = SWPADDENV(128)
```

See Also:

SWPADDSTR() / SWPADDSTRBAS() Return a string to a previous parent program

---

Function : **Return a string to a previous parent program**

Syntax   : **niValue = SWPADDSTR(cPID, cString)**

Return   : **A logical value**

SWPADDSTR() can be used to return a string of up to 127 bytes from the child program to a parent. The string to be returned may not contain any NULLs, - the first NULL will be interpreted as a string terminator.

The parent to receive the string is identified using the program ID of the parent program (set in the parent using SWPSETPID()). The string may be retrieved by the parent program using SWPGETSTR().

Example :

```
success := SWPADDSTR("TEST","OK")
```

returns the string "OK" to the program that has set a program ID of "TEST".

The return value is a logical indicating success or failure. There are three possible reasons for failure:

1 - the named parent program kernel is not currently in memory
2 - the string to be returned is longer than 127 bytes
3 - the parent was linked with a version of Blinker prior to 2.10

See Also:

Function : **Enable or disable saving of the current directory**

Syntax : **lValue = SWPCURDIR(lFlag)**

Return : **The previous setting**

Default : **Enabled**

Enable or disable the restoring of the current drive and directory following the swap. When enabled, the swap kernel will restore the default drive and current directory following the swap.

Function : **Enable or disable displaying of message after swap**

Syntax : **lValue = SWPDISMSG(lFlag)**

Return : **The previous setting**

Default : **Disabled**

Enable or disable the display of a 'Press any key to continue' message prior to reloading the parent process. The kernel will display the message and wait for a keystroke prior to reloading the parent if the setting is TRUE, otherwise no action will be taken.

Example:

```
* Display message and wait for keystroke after
* the DIR command has executed
SWPDISMSG(.T.)
SWPRUNCMD("DIR",0,"","")
```

See Also:

Function : **Limit the swapper to using EMS 3.2 calls**

Syntax : **lValue = SWPEMS320(lvalue)**

Return : **The previous setting**

Some offbeat/OEM EMS emulators can sometimes give problems with the swapper's use of EMS 4.0 level functions that appear to be improperly implemented in the affected drivers. This function can be used to force the swapper to only use EMS 3.2 functions.

Example:

```
SWPEMS320(.T.)    // use only EMS 3.2 calls.
```

This function will not be needed in most cases.

Function : **Get child process return code**

Syntax : **niValue = SWPERRLEV()**

Return : **Child program return code**

This function returns the DOS exit code of the child process. The exit code returned is the exit code of the last process which terminated prior to the command interpreter returning to the swap kernel, the value of which can be confirmed by running the program from a DOS batch file and testing for the ERRORLEVEL returned by the program. In the case that the command interpreter was the only process executed, the return code will be that of the command interpreter.

Example:

```
* Attempt to swap out, compile and link a program
SWPRUNCMD("Compile myprog -m",0,"","")
IF SWPERRLEV() <> 0 && compile failed
   ? "Compile failed..."
ELSE
   SWPRUNCMD("BLINKER @LINK.LNK",0,"","")
ENDIF
```

Function : **Get the swap system major error code**

Syntax   : **niValue = SWPERRMAJ()**

Return   : **Swap system major error code**

If an error occurs during the swap process itself, the SWPRUNCMD()

function will return false. SWPERRMAJ() returns the major error code, indicating the type of error that occurred.

Example:

```
* Attempt to swap out, compile and link a program
success = SWPRUNCMD("Compile myprog -m",0,"","")
IF .NOT. success
   ? "Swap failed..."
   ? "Major Error Number ",SWPERRMAJ()
   ? "Minor Error Number ",SWPERRMIN()
ELSE
   err_lev = SWPERRLEV()
   IF err_lev = 0
      SWPRUNCMD("BLINKER @myscr",0,"","")
   ELSE
      ?"Compile failed - errorlevel ",err_lev
   ENDIF
ENDIF
```

See Also:

Function : **Get the swap system minor error code**

Syntax : **niValue = SWPERRMIN()**

Return : **Minor error code**

If an error occurs during the swap process, the SWPRUNCMD() will return false. SWPERRMIN() returns the minor error code, indicating the type of error that occurred.

Example:

```
* Attempt to swap out, compile and link a program
success = SWPRUNCMD("Compile myprog -m",0,"","")
IF .NOT. success
   ? "Swap failed..."
   ? "Major Error Number ",SWPERRMAJ()
   ? "Minor Error Number ",SWPERRMIN()
ELSE
   err_lev = SWPERRLEV()
   IF err_lev = 0
      SWPRUNCMD("BLINKER @LINK.LNK",0,"","")
   ELSE
      ? "Compile failed - errorlevel ", err_lev
   ENDIF
ENDIF
```

See Also:

Function : **Limit use of EMS memory for swapping**

Syntax : **niValue = SWPFREEMS(niValue)**

Return : **The previous setting**

This function can be used to limit the amount of EMS used by the swap system for program image storage, so that the specified amount can be left free for the child program.

Please note that this function only restricts the amount of EMS used by SWPRUNCMD(), it does not cause any EMS in use by the application to be freed. The amount of EMS specified as a parameter to this function must already be free at the time SWPRUNCMD() is executed.

Example:

```
SWPUSEEMS(TRUE)      // Enable use of EMS (default) but leave
SWPFREEMS(512)    // at least 512 Kb free if possible
```

See Also:

Function : **Limit use of XMS memory for swapping**

Syntax : **niValue = SWPFREXMS(niValue)**

Return : **The previous setting**

This function can be used to limit the amount of XMS used by the swap system for program image storage, so that the specified amount can be left free for the child program.

Please note that this function only restricts the amount of XMS used by SWPRUNCMD(), it does not cause any XMS in use by the application to be freed. The amount of XMS specified as a parameter to this function must already be free at the time SWPRUNCMD() is executed.

Example:

```
SWPUSEXMS(TRUE)      // Enable use of XMS (default) but leave
SWPFREXMS(512)    // at least 512 Kb free if possible
```

See Also:

Function : **Enable / Disable waiting for a keypress before restoring the parent image.**

Syntax : **lValue = SWPGETKEY(lFlag)**

Return : **The previous setting**

Default : **Disabled**

This function enables / disables waiting for a keystroke by the swap system before reloading the parent process. The video mode will be restored after the key press (if saving of the video mode is enabled). This function can be used to pause with a graphics image displayed.

Example:

```
SWPGETKEY(.T.)    // Wait for keypress
SWPVIDMDE(.T.)    // Save video mode
SWPRUNCMD("SHOWIMG NICEONE.IMG",0,"","")
```

See Also:

Purpose : **Specify amount of EMS to be used for overlay caching**

Syntax : **BLINKER CACHE EMS <nuMax[%]>, <nuMinLeave[%]>**

Default : **100,0% (100 Kb maximum cache size)**

Example :
```
# Use a maximum of 50% of available EMS,
# but leave at least 1024 Kb available
BLINKER CACHE EMS 50%, 1024
```

or
```
# Use a maximum of 512 Kb EMS,
# but leave at least 25% of the total free
BLINKER CACHE EMS 512, 25%
```

This command specifies the amount of EMS (LIM version 4.0 and higher) memory to be used for the overlay cache.

Two numeric parameters are used to specify the amount of EMS memory to be allocated and the amount to be left for program use. Both parameters may be specified as either a number of kb, or as a percentage of the total available EMS. The second parameter always overrides the first; e.g. if the first parameter calls for 256K and the second requires 50% of EMS to be left free, then half of EMS will be left free and the cache will fit itself into the remaining memory.

Blinker will never allocate a cache that is larger than the total size of all the overlays contained within the program. Please see Appendix A 'MAP file output' for information on determining the optimum cache size.

See Also:

SWPGETPID() / SWPGETPIDBAS() Determine if a given program is a parent to the current program.

Function : **Determine if a given program is a parent to the current program.**

Syntax   : **lValue = SWPGETPID(cIdString)**

Return   : **Logical value indicating whether the passed string was set as a program ID by a parent process.**

Determine if cIdString was a parent (at any previous level) of the currently executing application. cIdString must be 31 characters or less, or the function will return a false value. The string must exactly match an Id set by a parent program using a call to SWPSETPID(). This can be useful in preventing two copies of the same application from being executed.

Example:

```
IdStr = "Test Program"
IF SWPGETPID(IdStr)
   ? IdStr + " is already executing!"
   QUIT
ENDIF
SWPSETPID(IdStr)
```

See Also:

SWPGETSTR() / SWPGETSTRBAS() Retrieve string sent from child process via SWPSETSTR()

Function : **Retrieve string sent from child process via SWPSETSTR()**

Syntax   : **cString = SWPGETSTR()**

Return   : **A string**

SWPGETSTR() is used within the parent program to retrieve the string set by the child program.

Example:

```
// Parent program
SWPSETPID("PARENT")
SWPRUNCMD("child",0,"","")
string = SWPGETSTR()
? "The string returned is :", string         // 'Hello'
* eof parent

// Child program
if SWPGETPID("PARENT")
   SWPSETSTR("PARENT","Hello")
endif
quit
* eof child
```

See Also:

SWPKEYBRD() / SWPKEYBRDBAS() Stuff keystrokes into the child process

---

Function : **Stuff keystrokes into the child process**

Syntax   : **niValue = SWPKEYBRD(cKeyString)**

Return   : **0 String parsed OK**
1 Error - mismatched quotes. There is an open quote in the input string
2 Error - mismatched '{'. There is a mismatched '{' or '}' (non-ASCII key delimiter).
nn Any other value indicates parsing failed at that position in the input string.

The SWPKEYBRD function allows the parent program to stuff the keyboard with a series of keystrokes prior to executing the child program. Use of this function will allow you complete control over the execution of the child program.

SWPKEYBRD() should work with most child programs that use the BIOS keyboard functions 0,1,10H and 11H to read keyboard input. Programs that access the keyboard hardware directly (such as MS Word) will not see stuffed keystrokes, and so cannot be controlled by this function.

The maximum number of keystrokes that can be stuffed is 250.

SWPKEYBRD Parameters

SWPKEYBRD takes a single string parameter made up of the following components:

- Embedded sub-strings
- Embedded sub-strings within quotes
- Special key combinations and keys that cannot be specified in ascii form, such as {pgup} and (alt-f1}

Embedded sub-strings will usually contain ASCII characters found on the keyboard, such as A-Z, a-z, 0-9, !@#$%&*() and so on. These characters are translated by SWPKEYBRD() into the keycodes associated with the keys that bear the character legend on a US keyboard.

It is possible for an embedded sub-string to contain characters that are not present on the US keyboard, such as international characters, box and line characters and so on. When SWPKEYBRD() encounters such a character, it generates a keystroke that is the equivalent <alt-nnn> keycode, where nnn is the ascii value of the character in question.

Quotes

Embedded substrings may start and end with either a single (') or a double (") quote, but they must match. This allows you to use either within the quoted text:

```
swpkeybrd("{enter}'he said "hello" twice'{enter}")
swpkeybrd('{enter}"Please press 'X' now!"{enter}')
```

Note that most compilers will use the first and last quote within the string as delimiters. The string you pass to SWPKEYBRD() must be within another set of quotes.

Non-ASCII Key combinations

In addition to quoted text, SWPKEYBRD() can also pass other keystrokes that cannot be represented by ASCII characters. Many of these keystrokes may be used in conjunction with a shift state. The three available shift states are:

shift, ctrl, and alt

When a shift state key is specified, it should precede the actual key definition, and be separated from it by a '-' character. eg:

{shift-f7}{ctrl-a}{alt-esc}

You cannot use more than one shift state at a time - the BIOS does not support this. If the child program uses multiple shift states simultaneously, such as <ctrl>+<alt>, <ctrl>+<shift>, you will not be able to control it using SWPKEYBRD(), as it either directly accesses the keyboard hardware, or it uses shift state information in the BIOS data area, which SWPKEYBRD() does not attempt to emulate.

The following keys are available:

```
{f1} - {f12}
{shift-f1} - {shift-f12}
{ctrl-f1} - {ctrl-f12}
{alt-f1} - {alt-f12}
{ctrl-a} - {ctrl-z}
{alt-a} - {alt-z}
{ctrl-0} - {ctrl-9}
{esc} {tab} {bksp} {enter}
{space} {ins} {del}
{ctrl-esc} {ctrl-tab} {ctrl-bksp} {ctrl-enter}
{ctrl-space} {ctrl-ins} {ctrl-del}
{ctrl-prtscr}
```

Note that {ctrl-prtscr} is the only flavor of print screen that the bios passes through, and so is the only one that can be stuffed.

```
{alt-esc} {alt-tab} {alt-bksp} {alt-enter} {alt-space}
{alt-ins} {alt-del}
```

Numeric Keypad Cursor Control Keys

Please note these keys are found on the numeric keypad.

```
{up} {down} {left} {right}
{home} {end} {pgup} {pgdn}
{ctrl-up} {ctrl-down} {ctrl-left} {ctrl-right} {ctrl-home} {ctrl-end}
{ctrl-pgup} {ctrl-pgdn}
{alt-up} {alt-down} {alt-left} {alt-right} {alt-home} {alt-end}
{alt-pgup} {alt-pgdn}
```

Extended Cursor Control Keys

These keys correspond to the separate gray keys found on 101/102 key extended keyboards. The * character indicates these gray keys:

```
{*ins} {*del} {*end} {*pgup} {*pgdn} {*home}
{*up} {*down} {*right} {*left}
{ctrl-*ins} {ctrl-*del} {ctrl-*end} {ctrl-*pgup} {ctrl-*pgdn} {ctrl-*home}
{ctrl-*up} {ctrl-*down} {ctrl-*right} {ctrl-*left}
{alt-*ins} {alt-*del} {alt-*end} {alt-*pgup} {alt-*pgdn} {alt-*home}
{alt-*up} {alt-*down} {alt-*right} {alt-*left}
```

Numeric Keypad Keys

Normally, when you specify a number (0-9) as part of an embedded string, the keystroke generated will be from the numeric keys along the top of the keyboard. If you require to stuff keystrokes from the numeric pad, use the following (the # indicates the keystroke is from the numeric pad):

```
{#0} {#1} {#2} {#3} {#4} {#5}
{#6} {#7} {#8} {#9} {#.}
```

```
{#/} {#*} {#-} {#+} {#enter}
{ctrl-#0} {ctrl-#1} {ctrl-#2} {ctrl-#3} {ctrl-#4}
{ctrl-#5} {ctrl-#6} {ctrl-#7} {ctrl-#8} {ctrl-#9}
{ctrl-#.} {ctrl-#/} {ctrl-#*} {ctrl-#-} {ctrl-#+} {ctrl-#enter}
{alt-#0} {alt-#1} {alt-#2} {alt-#3} {alt-#4}
{alt-#5} {alt-#6} {alt-#7} {alt-#8} {alt-#9}
{alt-#.} {alt-#/} {alt-#*} {alt-#-} {alt-#+} {alt-#enter}
```

You may separate strings and non-ASCII key combinations with spaces. eg:

```
x = SWPKEYBRD(" <tab> 'Testing...' <enter>" )
```

International Keyboard Drivers

Keystrokes supplied to the child process are passed directly from the swap kernel, bypassing any international keyboard driver that may be loaded. This has the effect of making the keys appear to have come from a US keyboard. The international keyboard handler will regain control, as soon as SWPKEYBRD() has stuffed all its keystrokes.

Keystroke Translation

There are two different types of BIOS call that a program may use to read the keyboard, one type is for 'standard' keystrokes, and the other is for 'extended keystrokes'. SWPKEYBRD() will automatically provide the correct keycode for the BIOS call used.

Clearing the keyboard buffer

By default, SWPKEYBRD() fools the child program into thinking that the keyboard buffer has been cleared between each keystroke. This will make key stuffing operate more reliably with programs that clear the keyboard buffer on a regular basis (and would otherwise cause subsequent stuffed keystrokes to be lost). This behaviour can be changed using the function SWPKEYCLR().

How SWPKEYBRD() works

The string that is passed to SWPKEYBRD() is immediately translated into the equivalent sequence of keystrokes. When SWPRUNCMD() is called to execute the child program, the swap kernel (that part of SWPRUNCMD() that remains in memory during the swap), takes over the BIOS keyboard interrupt, and feeds these keycodes into the child program in response to BIOS calls made by the child program (or DOS), as if they were coming directly from the keyboard. Once all keys have been stuffed, the kernel allows normal keyboard input to be passed through to the child program.

Most programs that use either DOS or BIOS function calls to read keyboad input should be controllable by SWPKEYBRD().

Creating a text file using a popular text editor

This example stuffs the first line, followed by the {enter} key, then the second line, followed by another {enter}. Next the file is saved by stuffing {alt-f} (file menu) and then 's' (save), finally the editor is exited with {alt-x}.

```
keystr := " 'Mary had a little lamb,' {enter} "
keystr := keystr + ' "It's fleece was ..." {enter}'
keystr := keystr + "{alt-f}'s'{alt-x}"
swpkeybrd(keystr)
swpruncmd("EDIT MARY",0,"","")
```

Entering data into a (fictitious) spreadsheet program

This example stuffs the keyboard movements, enters the data, recalculates, saves and quitts the spreadsheet.

```
keystr = "{down}{down}{right}{right}"
```

```
keystr = keystr + " '2.75'{enter}"
keystr = keystr + " '!' {enter} "
keystr = keystr + " '/s' {enter} '/x' {enter} "
SWPKEYBRD(keystr)
SWPRUNCMD("SS TEST",0,"","")
```

See Also:

Function : **Alter SWPKEYBRD() behaviour**

Syntax : **niValue = SWPKEYCLR(niClearCount)**

Return : **The previous setting.**

This function alters the number of times that SWPKEYBRD() clears the keyboard buffer between stuffing keystrokes into the child process.

Normally, SWPKEYBRD() indicates to the child program that the keyboard buffer is empty between stuffing each keystroke. This prevents loss of keystrokes if the child program tries to clear the keyboard buffer itself by discarding any pending keystrokes.

However, under multi-tasking systems, such as DesqView, the operating system itself polls the keyboard to detect idle time, and as a result may 'steal' the keyboard buffer clear signal that the swap kernel is trying to send to the child process. For this reason, the swap kernel sends not one, but 5 keyboard clear signals between each keystroke, to ensure the child program will see at least one of them.

If you seem to be losing keystrokes, increasing this value through the use of this function may help.

Example:

```
SWPKEYCLR(10)        // 10 clears between each key
```

See Also:

Function : **Prevent the user from rebooting the computer**

Syntax : **lValue = SWPNOBOOT(lFlag)**

Return : **The previous setting**

Default : **Disabled**

This function is used to enable or disable trapping of the <CTRL><ALT><DEL> keystroke sequence while the kernel is resident during the swap, preventing the user from soft rebooting while a program is swapped out. Please note that this mechanism does not function under DesqView or similar environments which hook the keyboard vector and look for this key sequence, since such software intercepts the keystroke before it gets to the swap kernel.

SWPRUNCMD() / SWPRUNCMDBAS() Execute a child process.

Function : **Execute a child process.**

Syntax : **lValue = SWPRUNCMD(cCommand, nuMem, cRunPath, cTempPath)**

Return : **TRUE if the command interpreter was executed.**
FALSE if an error occurred

This function executes a child process of the current program, freeing conventional memory occupied by the parent program. Any non conventional memory used by the parent will remain allocated.

The parameters to SWPRUNCMD() are as follows:

cCommand: The name of the program, batch file or DOS internal command to be executed, followed by any required parameters. An empty string will cause the command interpreter to be loaded, giving a DOS prompt.

nuMem: The amount of memory in kilobytes (kb) to be freed prior to executing the child process. If this parameter is passed as zero, as much memory as possible will be freed.

cRunPath: The drive and directory to make current prior to executing the child process.

cTempPath: The drive and directory where the swap function should create a disk swap file containing the image (or partial image) of the parent program. Note that XMS and EMS memory will be used in preference to a disk swap file if available provided their use has not been prohibited through the use of SWPUSEEMS() and SWPUSEXMS().

Note that this parameter may be overridden at runtime through the use of the SWAP or BLINKER environment variables. For example:

```
SET SWAP=F:\TEMP;
```

or

```
SET BLINKER=/SDF:\TEMP
```

will cause the temporary diskfile to be created in F:\TEMP, regardless of the value passed as a parameter to the SWPRUNCMD().

The swap function will always invoke the current command interpreter as specified in the COMSPEC environment variable, which in turn will execute cCommand. This allows the parent program to execute any internal or external DOS command, a batch file or any program which can be executed from the DOS prompt. The return code of the child process can be retrieved using the function SWPERRLEV().

The return value is a logical (boolean) value indicating the success of the SWPRUNCMD() call. Following an unsuccessful call to SWPRUNCMD(), the program can use SWPERRMAJ() and SWPERRMIN() to determine the nature of the error.

Example:

```
success = SWPRUNCMD("",0,"","")
```

Shell to DOS, freeing as much memory as possible, save swap file to current directory if required.

```
success = SWPRUNCMD("WIN",0,"C:\WINDOWS","F:\")
```

This command will execute Microsoft Windows, freeing as much memory as possible. The swap function will change drive and directory to C:\WINDOWS

prior to executing the WIN command. If a temporary disk swap file is required, it will be created on F:\

```
success = SWPRUNCMD("DIR /W",100,"","")
IF (!success)
   ? "Swap failed"
   majerr = SWPERRMAJ()
   minerr = SWPERRMIN()
   ? "Error codes :", majerr, minerr
ELSE
   ? "Swap succeeded"
   ? "Errorlevel :", SWPERRLEV()
ENDIF
```

Execute the DOS DIR command, first freeing 100KB, cretae the swap disk file to the current directory if required. On return, check the swap function return code. If the swap failed display the swap major and minor error codes, if it succeeded, display the return code of the child process.

```
success = SWPRUNCMD("RRUN ABC",0,"","")
IF (!success)
   ? "Swap failed"
      majerr = SWPERRMAJ()
      minerr = SWPERRMIN()
      ? "Error code :", majerr, minerr
   ELSE
   ? "Swap succeeded"
   ? "Errorlevel :",SWPERRLEV()
ENDIF
```

Execute the R & R report writer, checking for errors on return.

See Also:

SWPSETENV() **Set environment variables for the child process**

Function : **Set environment variables for the child process**

Syntax : **lsuccess = SWPSETENV(cString)**

Return : **True if cString < 512 bytes long, false otherwise**

SWPSETENV() allows the parent process to add, amend and delete environment strings in the child's environment. The parameter to SWPSETENV() is a single string containing all the variables to be set. Each variable is separated from the next by a delimiter, the character CHR(255). You do not need a CHR(255) at the end of the string. If you need a CHR(255) in the string, specify two in a row.

Example:

```
str := "PROMPT=<Type Exit to return><$p$g" + CHR(255) + ;
   "SWAPPED=TRUE"
SWPSETENV(str)
```

This example will change the prompt, and add (or change) the 'SWAPPED' environment variable.

```
str := "CLIPPER=" + CHR(255) + "BLINKER=/OP1"
SWPSETENV(str)
```

This example will DELETE the 'CLIPPER' environment variable, and add (or change) the 'BLINKER' environment variable.

Note that these environment changes are only set at the time the swap is executed, and apply only to the child's environment, NOT the parent's copy.

Use of this function will increase the size of the kernel in memory by an amount equal to the size of the new environment.

See Also:

SWPSETPID() / SWPSETPIDBAS() Set a program ID string for the current program.

Function : **Set a program ID string for the current program.**

Syntax   : **lValue = SWPSETPID(cIdString)**

Return   : **True if the ID was set**

Set the program ID of the current application to cIdString. The maximum length of cIdString is 31 characters. If the string length is greater than 31, the function will return false, and the program ID will not be set.

A program ID can always be set, regardless of whether a program at a higher level has set the same ID. A child process of the current program can determine whether a given ID has been set using the function SWPGETPID().

Example:

```
* Set the program ID to "Test Program"
IdStr = "Test Program"
SWPSETPID(IdStr)
```

See Also:

SWPSETSTR() / SWPSETSTRBAS() Set a string to return to the parent process.

Function : **Set a string to return to the parent process.**

Syntax   : **niValue = SWPSETSTR(cPID, cString)**

Return   : **A logical value**

SWPSETSTR() can be used to return a string of up to 127 bytes from the child program to a parent. The string to be returned may not contain any NULLs, - the first NULL will be interpreted as a string terminator.

The parent to receive the string is identified using the program ID of the parent program (set in the parent using SWPSETPID()). The string may be retrieved by the parent program using SWPGETSTR().

Example:

```
success := SWPSETSTR("TEST","OK")
```

returns the string "OK" to the program that has set a program ID of "TEST".

The return value is a logical indicating success or failure. There are three possible reasons for failure:

1 - the named parent program kernel is not currently in memory
2 - the string to be returned is longer than 127 bytes
3 - the parent was linked with a version of Blinker prior to 2.10

See Also:

Function : **Enable or disable use of EMS by the swap system.**

Syntax : **lValue = SWPUSEEMS(lFlag)**

Return : **The previous setting**

Default : **Enabled**

This function enables or disables use of EMS for swap image storage. The return value is the previous setting. If EMS use is enabled, the parent image can be saved (and restored) much more quickly that if it were written to disk.

Note that when both XMS and EMS memory are available, the swap system will use XMS first.

Note also that the BLINKER environment variable setting /SEc will override this function.

See Also:

SWPUSEUMB() **Enable or disable the use of UMBs by the swap system.**

Function : **Enable or disable the use of UMBs by the swap system.**

Syntax : **lValue = SWPUSEUMB(lFlag)**

Return : **The previous setting**

Default : **Enabled**

This function enables or disables use of UMBs (upper memory blocks) for the resident kernel. If available, and their use is enabled, the resident kernel which remains in memory during execution of the child process will relocate itself into a UMB, freeing up slightly more memory.

Note that the BLINKER environment variable setting /SUc will override this function.

See Also:

Function : **Enable or disable use of XMS memory by the swap system.**

Syntax : **lValue = SWPUSEXMS(lFlag)**

Return : **The previous setting**

Default : **Enabled**

This function enables or disables the use of XMS memory for swap image storage. The return value is the previous setting. If XMS use is enabled, the parent image can be saved (and restored) much more quickly that if it were written to disk.

Note that when both XMS and EMS memory are available, the swap system will use XMS first. Note also that the BLINKER environment variable setting /SXc will override this function.

See Also:

[Next Entry>>]

SWPVIDMDE() **Enable or disable restoring of the video page and mode after the swap**

Function : **Enable or disable restoring of the video page and mode after the swap**

Syntax : **lValue = SWPVIDMDE(lFlag)**

Return : **The previous setting**

Default : **Disabled**

When enabled, this function causes the swap kernel to save the video mode and active page prior to the swap, and restore them following the swap. Note that the contents of the video buffer are not saved.

1149: No longer used

Purpose : **Specify amount of XMS to be used for overlay caching**

Syntax : **BLINKER CACHE XMS <nuMax[%]>, <nuMinLeave[%]>**

Default : **100,0% (100 Kb maximum cache size)**

Example : **# Use a maximum of 50% of available XMS,**
 # but leave at least 1024 Kb available
BLINKER CACHE XMS 50%, 1024

or
 # Use a maximum of 512 Kb XMS,
 # but leave at least 25% of the total free
BLINKER CACHE XMS 512, 25%

This command specifies the amount of XMS (version 2.0 and higher) memory to be used for the overlay cache.

Two numeric parameters are used to specify the amount of XMS memory to be allocated and the amount to be left for program use. Both parameters may be specified as either a number of kb, or as a percentage of the total available XMS. The second parameter always overrides the first; e.g. if the first parameter calls for 256K and the second requires 50% of XMS to be left free, then half of XMS will be left free and the cache will fit itself into the remaining memory.

Blinker will never allocate a cache that is larger than the total size of all the overlays contained within the program. Please see Appendix A 'MAP file output' for information on determining the optimum cache size.

See Also:

[<<Previous Entry]

Error message overview

The following sections contain a complete list of the Blinker error messages with a description of each error and reasons for their occurrence.

The errors are divided into the following classes:

- 10xx - Link time warning messages
- 11xx - Link time fatal messages
- 12xx - DOS real mode runtime messages
- 13xx - DOS protected mode runtime messages

DOS Errorlevel

Link time warning messages are for information only, and do not set the DOS ERRORLEVEL, so the execution of Blinker and any MAKE utilities being used will continue unaffected. These warning messages may be disabled altogether using the BLINKER MESSAGE NOWARNING command.

Link time fatal messages and runtime messages, on the other hand, are of a more serious nature. They cause immediate termination of program execution, and set the DOS ERRORLEVEL to 1.

Blinker's Error Handler

At runtime when a program linked with Blinker encounters an error condition, it passes an error number to the Blinker error handler module. This routine supplies the text for the message which is to be displayed. The error handler is automatically included by Blinker in the .EXE in the course of linking.

The source for the Blinker error handler has been supplied on the Blinker distribution diskette in a number of languages, so the text of the Blinker error messages can be modified.

If you modify the Blinker error handler, you must then compile it and link it as a FILE into the root of the program. If you do not modify it, there is no need to compile it or include it in the link script. It will be automatically included into the .EXE by Blinker.

Care should be taken when overlaying the error handler in the case of the potential error 1201: "cannot find overlay file in the current path". This can cause an endless loop when the Blinker overlay manager cannot find an overlay file, tries to load the error handler, cannot find the overlay file containing the Blinker error handler, and again tries to load the error handler.

The developer may wish to handle this particular error in a different way to avoid this problem, or may just ensure that the overlay file can be found, e.g. by making it an internal overlay.

Note for CA-Clipper:

The error handler sets up the appropriate error message for the error number returned by the function BLIERRNUM(), in some cases using the function BLIERRPRM() to obtain the name of the offending item. In the case of CA-Clipper Summer '87 it then calls MISC_ERROR with a procedure name of BLINKER, a line number which is the error number, and the appropriate error message. In the case of CA-Clipper 5.x it creates a new error object which is passed as the argument to the CA-Clipper default error handler block.

Approaches to Debugging

Usually link time errors are well identified and the approach to their resolution is given in the individual error message descriptions in the

following section. Some additional techniques for discerning link time problems include using:

- The VERBOSE link script command, to see what Blinker is doing when an error occurs.
- The MAP command, to see how an .EXE has been constructed.
- The ECHO command, to output your own messages from within the link script.
- The command(s) BLINKER LINK EMS/XMS OFF, which disable the use of non-conventional memory at link time.

Runtime error conditions which arise from the linking process can be more difficult to diagnose and resolve.

The two most common link time mistakes which can lead to runtime errors are: (1) overlaying something which should not be overlaid, and (2) including the wrong module of two or more like-named modules. Both of these can lead to a variety of runtime symptoms which include, but are not limited to, corruption of variables or memory, inappropriate characters displayed to the terminal, a memory manager exception, and system lockups.

When encountering any such problems which do not appear to be attributable to the code itself or to other factors in the environment, the following steps are suggested in analyzing the linker's role in the problem.

1. Ensure that the program stack is sufficiently large using the STACK or BLINKER PROCEDURE DEPTH commands.

2. Verify that everything in the overlay area is reliably overlayable by (a) checking Blinker's 3RDPARTY.DOC file, (b) checking the documentation for each third party product for their recommended method for dynamic overlaying, and (c) if the problem is reproducible, removing all suspected files and libraries from the overlay area and testing the program after each is systematically reinserted.

3. Check for duplicate modules using the BLINKER MESSAGE DUPLICATES command. See Chapter 9 for the definition of this command. Usually this situation can be resolved by altering the order of the libraries and, if necessary, using the MODULE FROM command.

4. Test for non-conventional memory conflicts, using BLINKMEM.EXE, which is included with Blinker, and disable Blinker's runtime use of nonconventional memory either in the link script or via the BLINKER environment variable.

5. Perform a worst-case test of Blinker's overlaying process by using the MURPHY link-script command. For details on using this command, please see the appropriate section in Chapter 5, 7 and 9.

6. Attempt to isolate a small portion of the program which produces the error.

7. Attempt to test the program with another linker. If the problem does not occur, retry Blinker with the same overlays and order of libraries as the other linker.

8. Contact Blinker technical support. The technical bulletin section on our BBS and our FastFax retrieval system have our latest technical reports. Please refer to these before sending a BBS message or calling our technical support line. Also, review this manual for additional information, particularly the details in Chapters 3, 4, 7, and 9. If you should call, please have your Blinker serial number and link script files at hand.

Link Time Warning Messages

Error message numbers 1001 through 1099 are link time warning messages.

1001: ignoring invalid link script file command parameter in '<command>'

This message indicates that the link script file command specified has been ignored as it contains an invalid parameter value.

See Chapter 9, Blinker Commands for details on the parameters required by the command displayed.

1002: duplicate input file <filename> ignored

This message indicates that the same input file appeared in more than one FILE or LIBRARY command. The duplicated input file will be ignored.

1003: warning - no stack segment in .EXE file

This message indicates that the program .EXE file was created without a stack segment, which may cause the program to fail at runtime unless it is designed to set up its own stack in which case this message may be ignored.

1004: <filename(modulename)> : '<symbol>' duplicated in <filename(modulename)>

This message indicates that the symbol <symbol> defined as public in the first module is also defined as public in the second module.

This will occur if two symbols in separate .OBJ or .LIB files have the same name. The first one found is the one which is linked into the program, and the second one is ignored.

1004: DEFINE command : '<symbol>' duplicated in <filename(modulename)>

This message indicates that the symbol <symbol> defined as public in a DEFINE command is also defined as public in the second module.

This will occur if a symbol in an .OBJ or .LIB file has the same name as one in a DEFINE command, but the symbol in the .OBJ or .LIB file will be ignored.

1005: the default overlay class '<class>' contains the word 'DATA'

This message indicates that the default overlay class, taken from the class of the first segment in the first .OBJ being linked, contains the word 'DATA'. Blinker currently only overlays code, so this message is displayed as a warning in case a non-standard .OBJ file is being linked as the first .OBJ file.

If the class name displayed is not a code class then check that the first .OBJ being linked contains a valid code segment as the first segment, or place one of your own program .OBJ files as the first .OBJ file in the script file.

If the class name displayed is correct as the class of code to be overlaid then you may override this warning message by including the command OVERLAY CODE, <class> in the link script file.

1006: <filename>: padding truncated to fit within 64kb

This message is a warning that a procedure in the named file has a size close enough to the 64kb limit to mean that there is not enough space to add the full padding for incremental linking.

Blinker automatically truncates the padding to ensure that the 64kb is not exceeded, but it is more efficient for overlaying and is a good programming practice to keep procedure size down, typically to a few hundred lines.

1007: warning - program has no start address

This message indicates that no module containing a record indicating the starting point of the program was found during the link. A module of this type is usually found in the default language library for the compiler being used.

1008: <filename> : invalid incremental link file

This message indicates that the previous incremental link was unsuccessful and left an invalid incremental link file, or that the file specified with the BLINKER INCREMENTAL FILE command was not a valid incremental link file for the program concerned. A full link will be performed to recreate the program .EXE and its associated incremental link file.

1009: <filename> : file cannot be linked incrementally

This message applies to CA-Clipper applications only, and indicates that while attempting to incrementally link the application Blinker found that a non-CA-Clipper file had changed. Incremental linking of non-CA-Clipper files is not yet supported. A full link will be performed to recreate the application .EXE and its associated incremental link file.

1010: <filename> : no space available for incremental link

This message indicates that the named file has become too large to fit in the space reserved for it when the last full link was performed. A full link will be performed to recreate the application .EXE and its associated incremental link file.

If you receive this message on a regular basis you may wish to increase the space left for incremental linking by using the BLINKER INCREMENTAL PAD <padsize> command detailed in Chapter 9. Enlarging the padsize will increase the ratio of incremental links to full relinks, at the same time it will increase the size of the .EXE and its memory requirements during execution.

1011: '<symbol>' has been aliased to two different symbols

This message indicates that the named symbol has been aliased to more than one different symbol name.

The second alias symbol is ignored.

1012: segments in GROUP <groupname> have conflicting attributes

---

This message indicates that segments in the named group have different attribute settings in SEGMENTS directives in the module definition file. Segments in the same group are stored as one segment so must all have the same attributes.

The last setting found in the definition file will be the one used.

1013: ignoring invalid module definition file command parameter in '<command>'

This message indicates that the module definition file command specified has been ignored as it contains an invalid parameter value.

See Chapter 8, Windows Programs for details on the parameters required by the command displayed.

1014: ignoring duplicate module definition file command in '<command>'

This message indicates that the module definition file command specified has been ignored as a previous command of the same type has already been encountered and only one is allowed.

1015: ignoring duplicate export for symbol '<symbol>'

This message indicates that a duplicate export definition for the specified symbol has been encountered but ignored.

1016: ignoring duplicate import for symbol '<symbol>'

This message indicates that a duplicate import definition for the specified symbol has been encountered but ignored.

1017 : ignoring duplicate definition for segment <segment>

This message indicates that a duplicate segment definition for the specified segment name has been encountered but ignored.

1018 : unable to export symbol <symbol>

This message indicates that an export definition for the specified symbol has been encountered but the symbol does not exist in the program so the export definition was ignored.

Link Time Fatal Messages

Error message numbers 1101 through 1199 are link time fatal messages.

1101: terminated by user

This message indicates that the user interrupted Blinker while it was running by pressing the Ctrl-C or Ctrl-Break keys.

1102: no input files were specified

This message indicates that no FILE commands were found on the Blinker command line or in any of the script files specified on the command line. Blinker requires at least one FILE command to indicate which object file(s) is to be linked.

1103: '......' is not a valid link script command

This message indicates that the command shown is not a valid Blinker link script command. See Chapter 9, Blinker Commands for details of the valid Blinker commands and how to use them.

1103: '......' is not a valid module definition command

This message indicates that the command shown is not a valid module definition command. See Chapter 6, Windows Programs for details of the valid module definition commands and how to use them.

1104: unbalanced BEGINAREAs and ENDAREAs

This message indicates that a different number of BEGINAREA and ENDAREA commands was found in the script file(s), which implies an invalid overlay structure. Generally only one BEGINAREA and ENDAREA section is needed for Blinker's dynamic overlays. If an overlay structure is being carried over from a structured linker, this can be simplified by reducing the number of overlay sections.

BLINKER CLIPPER PAGE   Enable / Disable automatic CA-Clipper 5.x code paging

----

Purpose : **Enable / Disable automatic CA-Clipper 5.x code paging**

Syntax   : **BLINKER CLIPPER PAGE ON | OFF**

Default : **ON (if BLINKER INCREMENTAL OFF)**
OFF (if BLINKER INCREMENTAL ON)

This command can be used to disable the CA-Clipper 5.x dynamic paging system for CA-Clipper code. The only reason for ever wishing to do this is if a CA-Clipper 5.x program with no overlays is required, for example to create an installation program. The paging system is automatically disabled by Blinker when incremental linking is enabled, since the two options are mutually exclusive.

When linking with BLINKER INCREMENTAL OFF, the paging system is normally automatically enabled, unless this command is used.

It is highly recommended that the CA-Clipper 5.x paging system be used in all CA-Clipper 5.x programs, as it is often much faster than conventional dynamic overlaying.

Note: Use of the BLINKER CLIPPER PAGE OFF command will result in symbol-table compression and automatic overlaying of CA-Clipper code (as with BLINKER INCREMENTAL OFF), but the overlaying will be handled by Blinker's overlay manager rather than CA-Clipper's dynamic paging system.

See Also:

1105: maximum script file nesting depth exceeded

This message indicates that a script file name (@<filename>) was found inside another script file, which increased the total nesting level for script files to greater than the maximum permissible 5 levels.

It may be possible to alleviate the problem by splitting the script files into two or more files specified one after the other on the Blinker command line.

For example:

```
BLINKER @script1 @script2
```

1106: not enough real memory to link

This message indicates that Blinker could not allocate enough real (DOS) memory to continue the link.

Blinker has a link time virtual memory system that can use EMS (3.2 and higher), XMS (2.0 and higher) memory (or disk space) when conventional DOS memory is fully allocated. However, Blinker still requires a certain amount of conventional DOS memory to complete the link cycle.

To make more memory available, either remove one or more TSRs currently resident, run Blinker from outside a make utility or reduce the number of buffers specified in the CONFIG.SYS. Reducing the number of SECTION INTO and SEARCH commands will also reduce the memory needed at link time.

1107: disk full writing output file

This message indicates that the destination disk is full. There was an error writing to the disk, possibly because write access has been denied or the output file name is invalid. If there is plenty of space available, then this message indicates that there may be a problem writing to the disk, which can be checked by attempting to copy some other files to the destination disk and path. An externally overlaid program will use less temporary disk space at link time than an internally overlaid program.

1108: stack overflow during linking

This message indicates that the runtime stack for BLINKER.EXE has overflowed and should not be displayed under normal circumstances. If this message is displayed, please reinstall the Blinker software, and if the error persists please contact Blink, Inc. technical support with details of your Blinker serial number and system configuration.

1109: unable to open file <filename>

This message indicates that Blinker was unable to find the specified file in the current directory where it was expected to be, so the full directory path for the file should be specified. If no file extension is specified for script files, Blinker assumes an extension of .LNK.

If the file has been specified in the link script with a FILE command, then check the OBJ environment variable. If it has been specified with a LIBRARY or SEARCH command, check the LIB environment variable.

This error can also occur when Blinker is attempting to create a swap file and the TEMP environment variable points to an invalid directory. When running on a network, it is also possible that another user deleted the file or that you do not have network access rights to the file.

1110: <file>: invalid object file or library file

This message indicates that the file shown, which may be an .OBJ file or a .LIB file, contains an invalid object record.

If the file is expected to be valid, then it has probably become corrupt. To rectify this either recreate the file by recompiling, or reinstall the appropriate file from the installation diskette.

1111: <filename(modulename)> : invalid segment definition

This message indicates that the file shown, which may be an .OBJ file or a .LIB file, contains an invalid object record.

If the file is expected to be valid, then it has probably become corrupt. To rectify this either recreate the file by recompiling, or reinstall the appropriate file from the installation diskette.

1112: <filename(modulename)> : invalid group definition

This message indicates that the file shown, which may be an .OBJ file or a .LIB file, contains an invalid object record.

If the file is expected to be valid, then it has probably become corrupt. To rectify this either recreate the file by recompiling, or reinstall the appropriate file from the installation diskette.

1113: segment <name> or its group exceeds 64kb

This message indicates one of a number of situations :

Two related libraries may have been separated by other libraries in the link script file, so that the code in one library cannot reach the code in the other library. This may also occur when related .OBJ files, possibly extracted from the same library, have been separated by too many other .OBJ files in the script file.

Non-overlayable code may have been overlaid. When third party products are linked into the program, please refer to the file 3RDPARTY.DOC supplied on the distribution diskette and to the products' documentation for their suggested method for dynamic overlaying.

Two libraries may have declared the same segment names in different classes, which should not occur in most programs.

Two or more libraries may have been specified in an incorrect order within the link script. See the section, "Ordering the Libraries" in chapter 4 of this manual.

The program's default data segment (usually named DGROUP in the program's MAP file) may have exceeded 64K in size. Try changing near data to far data, by declaring large data items as far, or instruct the compiler to do this automatically, e.g. using the Microsoft C/C++ -Gt compile switch.If using Borland C/C++, try changing the large memory model compile option to huge. If using CA-Clipper, determine which 3rd-party product is using the most of DGROUP by systematically removing each library and running the program with the //INFO parameter of the CLIPPER environment variable enabled. When the offending library is found, please contact that product's technical support.

If these approaches do not work, and recreating or reinstalling the specified file does not help, then please contact Blinker technical support with exact details of the problem.

CA-Clipper note:

For CA-Clipper applications, the usual cause of this problem is that the first .OBJ file specified on the command line or in the script file is not a CA-Clipper .OBJ file. When linking a CA-Clipper application the first .OBJ file specified must be the CA-Clipper starting program.

If the first .OBJ file is definitely a CA-Clipper .OBJ then please review Chapter 4 which discusses the order of third party libraries in the script file.

1114: <file(module)> : '<symbol>' duplicated in <file(module)>

The same information applies as for warning error message number 1004, but because of the nature of the duplication, linking cannot continue. To resolve this error, give one of the symbols a different name and recompile the appropriate .OBJ file.

1114: DEFINE command : '<symbol>' duplicated in <file(module)>

The same information applies as for warning error message number 1004, but because of the nature of the duplication, linking cannot continue. To resolve this error, either remove the DEFINE command or give the symbol a different name and recompile the appropriate .OBJ file.

1115: <file(module)> : '<symbol>'unresolved external

This message indicates that the '<symbol>', defined as external in module (module) of file <file>, cannot be found in any of the other .OBJ files or libraries in the script file.

The usual cause of this problem is that an object or library file required by the program has been omitted from the script file.

Another possible cause of this message is that some libraries require the use of the SEARCH command, as their library cross reference records are not complete. Try a SEARCH on the library you suspect contains the missing '<symbol>'. For a full description of the SEARCH command see Chapter 9.

If you are aware of the cause for this message and want to continue linking to create an executable file, use the command BLINKER EXECUTABLE NODELETE.

Note for CA-Clipper Summer '87:

If you get '__dos_hold' as an unresolved external then please check that you have the updated version of CLIPPER.EXE, CLIPPER.LIB and EXTEND.LIB from Computer Associates (Summer '87). The original versions all have a date of 12-21-87 and a time of 12:00 am (which may not be displayed), while the updated versions have the same date, but a time of 2:00am. Blinker will work correctly only with the updated versions, as there were a number of problems fixed in them.

1116: fixup overflow at offset <offset> in segment <segment> contained in <module> referencing <symbol> co

----

This message indicates the exact point in a program which makes a near reference to a symbol more than 64 Kb away. This may occur in one of a number of situations:

Two related libraries may have been separated by other libraries in the link script file, so that the code in one library cannot reach the code in the other library. This may also occur when related .OBJ files, possibly extracted from the same library, have been separated by too many other .OBJ files in the script file.

Non-overlayable code may have been overlaid. When third party products are linked into the program, please refer to the file 3RDPARTY.DOC supplied on the distribution diskette and to the products' documentation for their suggested method for dynamic overlaying.

Two libraries may have declared the same segment names in different classes, which should not occur in most programs.

Two or more libraries may have been specified in an incorrect order within the link script. See the section, "Ordering the Libraries" in chapter 4 of this manual.

If these solutions do not resolve the problem then it will be necessary to modify the code so that <symbol> is accessed with a far reference, or ensure that the specified segment and symbol are accessible to one another by examining the MAP file for their respective locations. To determine which library is causing the problem, systematically remove each linbrary and relink.

1117: use of debugging information requires external overlays

---

External overlays must be created using the SECTION INTO command when adding debugging information to overlaid programs. For more information please see the section of chapter 7 which deals with Blinker and debuggers.

1118: too many external overlays

This message indicates that more than 15 unique external overlay file names were specified in the link script file.

This version of Blinker only supports up to 15 external overlays. It should be noted that with Blinker's dynamic overlaying, the only reason to have more than one external overlay file is so that each overlay file will fit onto a 360kb diskette, if required for distribution purposes.

For a full description of external overlays see the command SECTION INTO in Chapter 9.

1119: no main module found

This message indicates that no start-up procedure was found in the files and libraries to be linked. It usually indicates that the language library containing the start-up code has not been specified in the link script.

1120: incremental link file is too large

This message indicates that the incremental link file created by Blinker has exceeded the maximum allowable size.

The simplest way to reduce the size of the incremental link file is to place CA-Clipper .OBJ files which are rarely changed into a library file using a librarian such as MS LIB. This reduces the incremental link file size as incremental information is not stored for library files.

Alternatively, .CLP files can be used to create larger .OBJs, but the compile and link cycle would be slowed down because of the extra compile time.

1121: BASIC requires the first .OBJ file to be in the root

Microsoft BASIC requires the main .OBJ file of a program to be in the root, i.e. outside any BEGIN/ENDAREAs. All other BASIC .OBJs can be overlaid.

1122: <file(module)> : compiled with Summer '87 not 5.x

This message indicates that Blinker has determined that the specified module was compiled with CA-Clipper Summer '87, but that the first FILE in the application was compiled with CA-Clipper 5.x. All CA-Clipper compiled modules in a single application must be compiled with the same version of CA-Clipper.

Ensure you are linking the correct versions of the CA libraries and any third party libraries for the version of CA-Clipper that you are using, and also make sure all CA-Clipper compiled .OBJ files were compiled with the same version of CA-Clipper.

1123: <file(module)> : compiled with 5.x not Summer '87

This message indicates that Blinker has determined that the specified module was compiled with CA-Clipper 5.x, but that the first FILE in the application was compiled with CA-Clipper Summer '87. All CA-Clipper compiled modules in a single application must be compiled with the same version of CA-Clipper.

Ensure you are linking the correct versions of the CA-Clipper libraries and any third party libraries for the version of CA-Clipper that you are using, and also make sure all CA-Clipper-compiled .OBJ files were compiled with the same version of CA-Clipper.

1124: <file(module)> : first .OBJ not compiled with CA-Clipper

---

Within an application that contained a CA-Clipper module, the first module in a FILE command was not compiled with CA-Clipper. In a CA-Clipper application, the first FILE command must refer to the CA-Clipper starting program.

If you receive this message, reorganize your link script so that the first FILE statement refers to a CA-Clipper compiled module.

1125: <file(module)> communal '<symbol>' has an inconsistent element size

---

This message is applicable to C compiled modules. Blinker has detected that the size of the far static uninitialised data item (communal variable) indicated had previously been declared to have a different element size in a previously encountered module.

If you receive this message, modify and recompile your C modules so that the communal variable is declared to be the same size in all the modules.

1126: <file(module)> communal '<symbol>' has an invalid data type

---

This message is applicable to C compiled modules. Blinker has detected that the type of the indicated communal variable is neither NEAR nor FAR.

If you receive this message, it is likely that the indicated file has been corrupted. Recompile or reinstall the module.

1127: near communals segment exceeds 64kb

The total size of the near communal variables segment exceeds the maximum permissible size (64kb).

To reduce the size of the segment change some near uninitialised data declarations to far in the program source.

1128: <file(module)> : unrecognized .OBJ record type nnh

Blinker encountered an object record type which is not currently supported. It is possible that the module in question is corrupted.

If recreating or reinstalling the module does not solve the problem then please contact Blinker technical support with your serial number and a description of the problem.

Also, Blinker may have encountered a new type of object record, such as one created by a new compiler version not supported by the version of Blinker being used. If not using the latest version of Blinker, please enquire about upgrading.

1129: BLINKER OVERLAY DYNAMIC is only applicable to CA-Clipper Summer '87

Blinker supports the dynamic overlay allocation method (whereby memory for overlays is allocated from Clipper's free pool) for CA-Clipper Summer '87 only, and the application being linked is not a CA-Clipper Summer '87 application.

Remove the BLINKER OVERLAY DYNAMIC command from your link script.

1130: unable to create file <filename>

If this error should occur then it is likely that either there is already an existing file of the same name which has been assigned read only status, or no create access rights have been granted to the destination drive or directory, or that the destination disk is full.

1131: error closing file <filename>

If this error occurs then it is likely that the file specified has been inadvertently deleted by another user on the system while Blinker was still using it, or there could be a network or disk failure.

1132: error deleting file <filename>

This message indicates that either the file specified is marked as read only or is still in use, or that network delete access is not currently granted, or that the file exists in an alternate directory specified with the DOS APPEND command or network search command.

1133: version n.n is required to use this .LNK file

This message indicates that the .LNK file being used contains a request for a version of Blinker later than the version you are using.

1134: error reading file <filename>

Blinker was unable to read from the specified file. Please check to make sure the file is accessible via the appropriate PATH and other environment variables, or is specified with a full path in the link script. Alternatively, the file may be in use or corrupt.

1135: error writing file <filename>

Blinker was unable to write to the specified file. Check available disk space, and that an undeleteable file with the same name is not preventing Blinker from writing to the file.

1136: no more XMS handles available

This message indicates that there were no more XMS handles available for Blinker's linktime virtual memory. Please increase the number of handles provided by your XMS memory manager.

1137: no more EMS handles available

This message indicates that there were no more EMS handles available for Blinker's linktime virtual memory. Please increase the number of handles provided by your EMS memory manager.

1138: unable to find and execute program

The program CVPACK.EXE, required for use with CodeView version 4, could not be located after the link. Symantec C++ includes a similarly named utility which should be copied to CVPACK.EXE.

1139: unbalanced DEFBEGINs and DEFENDs

This message indicates that a different number of DEFBEGIN and DEFEND commands was found in the module definition file(s), which implies an invalid link script file. Please correct and retry.

BLINKER DEBUG HANDLE   **Specify output file handle for debugging routines**

Purpose : **Specify output file handle for debugging routines**

Syntax : **BLINKER DEBUG HANDLE <nihandle>**

Default : **1 (screen output)**

Example : **# Set default output handle to the printer**
BLINKER DEBUG HANDLE 4

This command sets the default handle used by those Blinker functions which optionally take a handle as a parameter.

They are detailed in Chapter 10, Blinker Functions. The handle number can be set to either 1 (the default) to send output to the screen, 2 to send output to the standard error device or 4 to send output to the printer.

See Also:

1140: new .EXE contains more than 254 segments

This message indicates that the new .EXE file would contain more than the maximum of 254 segments. This should only occur if the BLINKER SEGMENT THRESHOLD command is used to reduce the size of combined segments, so the THRESHOLD should be increased until the .EXE is created successfully.

1141: new .EXE needs larger alignment value

This message indicates that the new .EXE file would be larger than 65535 times the .EXE aligment value specified by the BLINKER EXECUTABLE ALIGNMENT nnn command. This value defaults to 16, and is always a power of two, but should be increased until the .EXE is created successfully.

1142: <filename(modulename)> : <segment> already belongs to group <group>

This message indicates that a segment in the specified module has been defined as being in two distinct groups, which is not permitted.

1143: <filename(modulename)> : unable to export symbol <symbol>

This message indicates that a request to export the specified symbol was made, but the symbol was either undefined or a constant. Exported symbols must be associated with a data or code segment.

1144: the size of DGROUP plus the heap exceeds 64 kb

This message indicates that the size of the heap, when added to the size of DGROUP including the stack, exceeds 64 kb.

The heap size may be set or reduced using the module definition file command HEAPSIZE nnnn, or may be set to the maximum using HEAPSIZE MAXVAL.

1145: symbol <symbol> uses duplicate ordinal <ordinal>

This message indicates that a request to export the specified symbol was made, but the ordinal number specifed for the export had already been used by another exported symbol.

1146: command APPLOAD must appear before SEGMENTS commands

This message indicates that an APPLOAD command was found in the module definition file after one or more SEGMENTS commands, which is not permitted.

1147: <filename(modulename)> contains incremental compilation errors

---

This message indicates that the specified .OBJ file was created with a compiler which supports incremental compilation, but that the compilation failed. Please correct and recompile the specified .OBJ file and then relink.

1148: BLINKER CLIPPER SYMBOL OFF required for huge symbol table

---

This message will only occur with CA-Clipper 5.x Windows programs, and indicates that the symbol table is too large to perform symbol table compression due to limitations in the CLIPPER.LIB run time.

Simply add the command BLINKER CLIPPER SYMBOL OFF to the end of the link script file to remove this error message and create the .EXE.

1150: <filename> is a demo so cannot be linked with this Blinker

---

This message indicates that the specified library is a special demonstration library which can only be linked with a demonstration version of Blinker. The demonstration version of Blinker should have been supplied along with the demonstration library.

1151: increase PACKDATA parameter to reduce SYMBOLs segments

This message indicates that there were too many SYMBOLs segments in an incrementally linked CA-Clipper program, so it is necessary to increase (or add) the parameter to the PACKDATA link script file command.

1152: too many exported name symbols

This message indicates that the .EXE or .DLL file being created contains too many exported name symbols to be placed in the .EXE header.

Either remove some EXPORT commands from the module definition file or remove one or more import libraries from the link script file.

1153: <n.n> requires a more recent version of BLXxxxnnn.LIB

This message indicates that the indicated library is from an older version of Blinker.

Check that the link script file and the LIB environment path point to the latest version of the indicated library.

1171: .EXE file size limit exceeded

This message indicates that the .EXE file exceeded Blinker's internal size limit. Please contact Blink, Inc technical support with your Blinker serial number and exact details of the circumstances causing the error.

1172: an undetermined XMS error has occurred

The 1172 error can be avoided temporarily by turning off Blinker's link time use of XMS with the BLINKER LINK XMS OFF command until the source of the XMS problem can be resolved. This assists in identifying the problem but is not recommended as a resolution.

1173: an undetermined EMS error has occurred

The 1173 error can be avoided temporarily by turning off Blinker's link time use of EMS with the BLINKER LINK EMS OFF command until the source of the EMS problem can be resolved. This assists in identifying the problem but is not recommended as a resolution.

1174: invalid file number

Errors 1174 and 1175 should not occur in normal operation. If you receive one of these messages please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1175: too many buffered files

Errors 1174 and 1175 should not occur in normal operation. If you receive one of these messages please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1176: invalid symbol token

Blinker has encountered an invalid CA-Clipper symbol token when compressing the symbol table of a CA-Clipper application. The most likely cause of this is corruption of one or more object modules or libraries.

Recompile your program and link with VERBOSE mode enabled in order to determine the module causing the error.

1177: invalid packed symbol number

Blinker has encountered a problem while compressing the symbol table of a CA-Clipper application. The most likely cause of this is corruption of one or more object modules or libraries.

Recompile your program and link with VERBOSE mode enabled in order to determine the module causing the error.

1178: invalid local symbol number

Blinker has encountered a problem while compressing the symbol table of a CA-Clipper application. The most likely cause of this is corruption of one or more object modules or libraries.

Recompile your program and link with VERBOSE mode enabled in order to determine the module causing the error.

This can occur in a CA-Clipper application when the CA-Clipper 5.x libraries have been renamed and the CA-Clipper Summer '87 libraries are included automatically as default libraries. In this case, use the command NODEFLIB.

1179: too many entries in packed symbol table

Blinker has reached an internal size limit while compressing the symbol table of a CA-Clipper application. This is extremely unlikely to occur in normal operation.

Please contact Blinker technical support with your serial number and exact details of the circumstances causing this error.

1180: invalid communal number

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1181: too many entries in LNAMEs table

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1182: too many entries in sort table

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1183: too many entries in symbol names table

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1184: too many entries in dynamic table

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1185: dynamic table space is full

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1186: too many fixups in overlay

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

BLINKER DEMONSTRATION DATE   Specify expiration date for demonstration version

----

Purpose : **Specify expiration date for demonstration version**

Syntax   : **BLINKER DEMONSTRATION DATE <yyyy/mm/dd>**

Example : **# Set the program expiration date**
BLINKER DEMONSTRATION DATE 1992/12/31

This command is used to set the expiration date of a program, after which it will end with the "demonstration date limit exceeded" message. It may be set to any date in the format yyyy/mm/dd.

This can be used to create a demonstration version of the program without having to modify the source code. When the overlay manager loads overlays after the expiration date (as determined by the DOS system date), the error handling procedure BLERRPRG is called with error number 1208, indicating that the demonstration date limit has been exceeded. The source for this error handler is provided should you want to modify the text of the termination message.

Note: The demonstration restrictions are only checked when overlays are called, so to use them please ensure that at least one function or procedure which is called regularly is overlaid. To be sure thatoverlays exist, be aware that any overlaid segment that is smaller than the BLINKER OVERLAY THRESHOLD default of 32 bytes will automatically be removed from the overlay area.

CA-Clipper 5.x note:

To include Blinker's overlay manager in the application, some non-CA-Clipper code must be overlaid, for example EXTEND.LIB.

See Also:

1187: not enough virtual memory to link

Errors 1180-87 indicate a Blinker internal size limit has been exceeded for the specified table. If you receive one of these messages, first upgrade your version of Blinker to the most current for the largest internal table size. If upgrading does not remedy the problem, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1188: .EXE file is corrupt

This message indicates that the BLINKER.EXE file has become corrupted and should be reinstalled from the distribution diskette.

1189: invalid page status

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1190: internal error - invalid symbol number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1191: internal error - invalid library number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1192: internal error - invalid page number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1193: internal error - invalid CodeView number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

[^^Up^^]

1194: internal error - invalid module number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1195: internal error - invalid public number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1196: internal error - invalid external number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1197: internal error - invalid group number

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1198: internal error - invalid segment number

Error messages 1189-1199 should not occur during normal operation.

The 1198 error can be caused by linking .OBJs which have been compiled with Borland C's overlay options 'Y' and 'Yo.' These options should only be used when linking with TLink.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

1199: internal error - DOS function call error

Error messages 1189-1199 should not occur during normal operation.

One of the most common causes within this group of errors is a memory conflict. Run the BLINKMEM utility, which is included with Blinker, to check for such conflicts.

Another common cause of these errors is a corrupted object file or library. Re-run the link with the keyword VERBOSE added: the last module or library listed (or, in some cases, the module or library after the last one listed, in the sequence of the link script) will most likely be the offending file. Rebuild or reinstall the file.

If none of these procedures resolve the error, please contact Blinker technical support with your serial number and exact details of the circumstances causing the error.

DOS Real Mode Run Time Error Messages

Error message numbers 1201 through 1299 are real mode run time messages.

1201: unable to find overlay file <filename> in current PATH

This message indicates that either too many files are open or that the external overlay file <filename>, specified and created at link time, can no longer be found in the current directory or the current PATH.

Ensure that the overlay file is in the current directory, the directory containing the .EXE file, or in a directory in the DOS PATH. If so then try marking the .EXE and overlay file(s) read-only

Note: The Blinker errors 1201 through 1204 can occasionally result from the effects of a virus, so if the above measures do not help then perform an extensive check for viruses and take the necessary measures for removal.

1202: DOS read error in file <filename>

This message indicates that DOS returned an error while reading an overlay file.

A number of situations can cause this error to occur frequently and consistently:

(1) The most common is the use of a 'file handles' routine within the program to increase the maximum number of file handles available to the program under DOS versions 3.0 to 3.2. If this 'file handle' routine is called after Blinker has opened the overlay file and loaded one or more overlays from the file, the file handle table is recreated, wiping out Blinker's existing file handle for the overlays. To avoid this, ensure that the handles function is called from the root before Blinker has loaded any overlays.

CA-Clipper note:

A combination of three factors can cause this to occur in a CA-Clipper application: (a) Calling one of these handle table expanding routines such as FUNCKY's SETHANDLES() or NETLIB's NHANDLES() from an overlay, (b) Using versions of DOS 3.3 or above, and (c) specifying the CLIPPER Fn environment parameter. Since the handle-increasing routine is only needed for DOS versions 3.0 to 3.2, this error can be avoided by checking the DOS version before calling that routine:

```
   IF DOSVERSION () < 3.3
      SETHANDLES (29) # a FUNCky, NetLib, or shareware function
   ENDIF
```

(2) This error can also occur when non-overlayable modules are placed in the overlay area. If there are any .OBJ or .LIB files in the overlay area which are not known to be overlayable, then these should be moved to the root and the program relinked.

If the program then runs successfully, this indicates that the file(s) are not currently overlayable (see the file 3RDPARTY.DOC on the distribution diskette for an up-to-date list of libraries which are known to be overlayable, or call our Technical Support BBS to download the latest version of 3RDPARTY.DOC).

(3) Should the program stack overflow, it can overwrite the nearby file handle area, resulting in this error. Try increasing the size of the stack with either the STACK or BLINKER PROCEDURE DEPTH command.

(4) A network could be configured to deny access to the .EXE once it had been loaded, preventing Blinker's overlay manager from reading an internal overlay. Also swap functions can result in this error if the file handle table has not been restored correctly.

(5) If this error occurs sporadically, it is possible that the overlay file has been closed by an external influence such as a TSR or being logged off a network, or the overlay file has become corrupt because of a disk error and should be reinstalled.

Note: The Blinker errors 1201 through 1204 can occasionally result from the effects of a virus, so if the above measures do not help then perform an extensive check for viruses and take the necessary measures for removal.

1203: file <filename> is not a valid .EXE file

This message indicates that the specified file does not have a valid .EXE file header, thus the Blinker internal overlay cannot be located within the file. This indicates that the file has become overwritten by another file, that the PATH has changed so that an alternative file of the same name is found instead, or that the .EXE file has become corrupt.

Try recreating the .EXE file, using external overlays (via the SECTION INTO xxx command), and marking the .EXE and .OVL file(s) read-only.

Note: The Blinker errors 1201 through 1204 can occasionally result from the effects of a virus, so if the above measures do not help then perform an extensive check for viruses and take the necessary measures for removal.

1204: overlay file <filename> does not match the .EXE file

This message indicates that when the overlay manager opened the overlay file specified and created at link time, it was not a Blinker overlay file or it was not prepared at the same time as the currently executing .EXE file.

This indicates that the correct copy of the overlay file has become overwritten by an older version, the newer version cannot be found in the current PATH, or the previous link was terminated before the overlay file was created correctly.

Try recreating the .EXE file and marking the .EXE and .OVL file(s) read-only.

Note: The Blinker errors 1201 through 1204 can occasionally result from the effects of a virus, so if the above measures do not help then perform an extensive check for viruses and take the necessary measures for removal.

1205: not enough memory to load procedure

This message should only occur when using CA-Clipper Summer '87 with the dynamic method of overlay allocation (the default method for Summer '87). This message indicates that there is no more dynamic memory available in CA-Clipper's free pool to load the requested overlaid procedure.

For a description of how to maximize the amount of dynamic memory available to CA-Clipper please refer to Chapter 8.

1206: maximum procedure nesting depth exceeded

This message indicates that during execution of the program the depth

of nested procedures, i.e. the number of procedures calling one another without returning, has exceeded the maximum that may be handled on the procedure call stack.

To increase the size of the procedure-call-stack simply use a BLINKER PROCEDURE DEPTH or a STACK command in the Blinker script file.

Note: The errors 1207 through 1209 are NORMAL termination messages and simply indicate that a demonstration limit has been exceeded.

1207: No longer used

This message is no longer used.

1208: demonstration date limit exceeded

This message indicates that the program was linked with the BLINKER DEMONSTRATION DATE <limit> command to create a demonstration program with an expiration date, and that date has now been exceeded. For more details on demonstration versions see the related section in Chapter 7.

1209: demonstration time limit exceeded

This message indicates that the program was linked with the BLINKER DEMONSTRATION MINUTES <limit> command to create a demonstration program with a limited execution time, and that limit has now been reached. For more details on demonstration versions see the related section in Chapter 7.

1210: overlay has been prematurely freed

This message will only occur when using CA-Clipper Summer '87 and the dynamic allocation method for overlays. This message indicates that the memory assigned to an overlay has been prematurely freed by the CA-Clipper memory management routines, which can happen under very tight memory conditions.

This is a CA-Clipper Summer '87 error which the Blinker overlay manager checks for whenever it manipulates dynamic memory.

Do not confuse this message with the CA-Clipper 5.x internal error message 1210.

1211: overlay manager stack overflow

The Blinker overlay manager maintains a reload stack used in the management of dynamic overlays, and this stack has overflowed. The most common cause of this error is uncontrolled or unintentional recursion within an overlaid procedure or function, where a particular procedure or function is calling itself repeatedly.

Please increase your BLINKER PROCEDURE DEPTH when it is low, and check your code for unintentional recursion.

1212: Overlay Opsize exceeded - increase Opsize

This message applies only to the BLINKER OVERLAY FIXED overlay allocation method, and should not occur in normal operation.

Normally, Blinker calculates the minimum required opsize at link time, and does not allow the opsize to be set below this minimum size (if you set the opsize to a value smaller than the minimum, Blinker will use the minimum instead).

This message indicates that the Blinker overlay manager could not find space in the overlay area to load an overlay (the opsize was exceeded). Increasing the overlay opsize by 1-2kb, and relinking the program should solve the problem, but if you receive this message, please call Blinker technical support with your serial number and full details of the circumstances leading to the problem.

1213: attempt to call DEFINED routine

The link script command has been used to 'stub out' a particular routine, but a call to the routine has been made during execution of the program. Either remove all calls to the defined routine from the program, or remove the DEFINE command from your link script.

1214: error accessing EMS overlay cache

This message indicates that an error has occurred while the Blinker overlay manager is accessing EMS for the storage and / or retrieval of overlays in the overlay cache. If the problem persists then disable the BLINKER CACHE EMS command in the link script file.

If this fixes the problem then it is possible that you have a TSR or device driver working in the background which is manipulating EMS directly in a non-standard manner. Try removing all TSRs and device drivers from the system except for the EMS driver, and test if the problem still occurs . If it does then please contact Blinker technical support with your Blinker serial number, your system configuration and a description of the problem.

1215: error accessing XMS overlay cache

This message indicates that an error has occurred while the Blinker overlay manager is accessing XMS for the storage and / or retrieval of overlays in the overlay cache. If the problem persists then disable the BLINKER CACHE XMS command in the link script file.

If this fixes the problem then it is possible that you have a TSR or device driver working in the background which is manipulating XMS directly in a non-standard manner. Try removing all TSRs and device drivers from the system except for the XMS driver, and test if the problem still occurs . If it does then please contact Blinker technical support with your Blinker serial number, your system configuration and a description of the problem.

1216: overlay manager unable to resume

This message indicates that a call to the overlay manager resume function BLIOVLRES() has been made, and that the overlay manager was unable to re-allocate memory for the overlay pool in order to resume executing overlays.

The probable cause of this message is that subsequent to the call to the overlay manager suspend function, BLIOVLSUS(), the memory previously used for the overlay pool was allocated to a request for conventional memory and still remains allocated. In order to resume, the overlay manager requires an area of conventional memory at least as large as the previous overlay pool.

1217: overlay vector corrupted during execution

This message indicates that the overlay manager has detected an invalid overlay vector. The most likely cause of this is corruption at runtime due to an invalid pointer or attempting to overlay a non-overlayable module.

Another possible cause is the use of a debugger to place breakpoints in the code after an overlay call, since this replaces the vector data with a debugger breakpoint.

Use of the MURPHY command for debugging overlays may help to identify whether non-overlayable modules are responsible for this message.

1220: swap error [major/minor] (text message)

This message indicates that a fatal error has occurred in the Blinker swap system. The error consists of a major error number and an associated minor error number [major/minor] and a text message indicating the type of error that occurred.

Normally such errors are reported through the swap function error functions SWPERRMAJ() and SWPERRMIN(), however, in the case that the program cannot be restored to memory, a 1220 error will be output, and the program will return to DOS.

The following major error codes have the following meanings and text messages:

0	No error
1	Disk full
2*	File not found
3*	Disk I/O error
4	Change directory error
5	Temporary file creation error
6	Change drive error
7*	EMS error
8*	XMS error
9	SHELL error
10	Handle table too large to save
11+	Top of memory error
12+	MCB chain corrupt
13	Too many MCBs
14*	DOS memory function call error
15+	Unable to locate same segment
16	Unable to find COMSPEC in environment
254+	Swap internal error
255+	Swap internal error

Errors marked with a (+) are always fatal and will result in a 1220 error message. Errors marked with a (*) may result in a 1220 message, or may be non-fatal depending on when the error occurs in the swap process.

Errors that are not marked with either (+) or (*) are non-fatal (and will not result in a 1220 message), but the child process will not have been executed. These errors can be detected through the SWPERRMAJ() and SWPERRMIN() functions (see Chapter 10).

The explanation for each major error is as follows:

0 - No error

A major error of zero indicates that the swap function successfully restored the command interpreter, and restored the program image. Normally a minor error code of zero will be returned. In the case that the swap function had to remove a TSR from memory, however, the minor error code will be set to 1, purely for informational purposes.

If the size of DOS memory increased during the swap (by executing a program such as Quarterdeck's VIDRAM) the minor error code returned will be 2. Users should be STRONGLY discouraged from executing such programs during the swap. Please note that a decrease in DOS memory through the use of such a utility will result in a fatal error (major code 11).

1 - Disk full

The disk on which the swap function was attempting to save the program image is full.

The minor error code is the associated DOS error code.

2 - File not found

DOS returned a 'file not found' error when the swap function attempted to restore the program image from a disk swap file. The swap file may have been deleted.

The minor error code is the associated DOS error code.

3 - Disk I/O error

DOS returned an unexpected error while the swap function was either reading from or writing to the disk file containing the program image.

The minor code is the associated DOS error code.

4 - Change directory error

The swap function encountered an error either when changing to the requested directory passed as a parameter to the swap function or when restoring the current directory following the swap.

The minor code is the associated DOS error code.

5 - Temporary file creation error

DOS returned an error when the swap function attempted to create a temporary file to restore the program image. Check that the path specified for the temporary file is valid.

The minor code is the associated DOS error code.

6 - Change drive error

The swap function encountered an error either when changing to the requested drive passed as a parameter to the swap function, or when restoring the current drive following the swap.

The minor code is the associated DOS error code.

7 - EMS error

An unexpected error occurred during an EMS operation.

The minor code is the EMS function call that failed.

8 - XMS error

An unexpected error occurred during an XMS operation.

The minor code is the XMS function call that failed.

9 - SHELL error

The swap function was unable to execute the command processor. Check that the COMSPEC environment variable was set to point to the command interpreter.

The minor code is the associated DOS error code.

10 - Handle table too large to save

The swap function can only save an expanded handle table that contains 255 (or fewer) handles.

The minor error code is the number of handles in the current handle table.

11 - Top of memory error

The top of DOS memory was at a lower address following the swap than it was before the swap. Do not execute programs like Quarterdeck's VIDRAM utility while a program is swapped out.

The minor error code is the segment address at the top of memory.

12 - MCB chain corrupt

The swap function detected that the DOS memory control block chain was corrupted.

The minor error code is the segment address of the corrupted MCB.

13 - Too many MCBs

The swap function can only save programs that own 128 or fewer DOS memory control blocks, which is more than sufficient in most cases.

The minor error code is the number of MCBs.

14 - DOS memory function call error

DOS returned an error on a memory-related function call. Please contact technical support with full details of the situation causing the error.

The minor code is the associated DOS error code.

15 - Unable to allocate same segment

In order to restore the program image correctly, the swap function must be able to allocate the same memory blocks that the program originally owned. This message indicates that DOS returned a block at a different address than the swap function expected. The program cannot be correctly restored.

The minor error code is the segment address returned by DOS.

16 - Unable to find COMSPEC in environment

The swap function needs to locate the command interpreter (usually COMMAND.COM) in order to execute the child process. This error indicates that the COMSPEC environment variable that specifies the location of the command interpreter could not be found. Make sure you have a valid COMSPEC setting in the environment.

254 - Swap internal error

Please contact technical support with full details of the situation causing this error.

255 - Swap internal error

Please contact technical support with full details of the situation causing this error.

BLINKER DEMONSTRATION MINUTES Specify number of minutes for demonstration version

----

Purpose : **Specify number of minutes for demonstration version**

Syntax : **BLINKER DEMONSTRATION MINUTES <nuLimit>**

Example : **# Set the limit on length of program execution**
BLINKER DEMONSTRATION MINUTES 20

This command is used to set the number of minutes of program runtime before it will end with the 'demonstration minutes limit exceeded' message. It may be set to any number from 1 minute to 65535 minutes.

This can be used to create a demonstration version of the program without having to modify the source code. When the specified time limit is reached, the error handling procedure BLERRPRG is called with error number 1209, indicating that the demonstration minutes limit has been exceeded.

The source for this error handler is provided should you want to modify the text of the termination message.

Note: The demonstration restrictions are only checked when overlays are loaded, so to use them please ensure that at least one function or procedure which is called regularly is overlaid. To be sure that overlays exist, be aware that any overlaid segment that is smaller than the BLINKER OVERLAY THRESHOLD default of 32 bytes will automatically be removed from the overlay area.

The function BLIDEMMIN() allows the limit to be extended from within the program. Please see the appropriate section in Chapter 10 for more details.

CA-Clipper 5.x note:

To include Blinker's overlay manager in the application, some non-CA-Clipper code must be overlaid, for example EXTEND.LIB.

See Also:

DOS Protected Mode Run Time Error Messages

Error message numbers 1301 through 1399 are protected mode run time messages

1301: unable to load file <filename>

This message indicates that the DOS extender was unable to load the specified file. This is most likely to be caused by the file having been overwritten or corrupted, possibly due to an aborted link while creating the file. Please attempt to reinstall or recreate the file concerned.

It may also be caused by the current user having sufficient network access rights to open the file but insufficient rights to load and execute it.

1302: DPMI function call nnnnh failed with error code nnh

This message indicates that the specified error code was returned from the specified DPMI function call.

Please consult the DPMI documentation for further information on the meaning of the function call and the error code.

1303: DOS function call nnh is not supported

This message indicates that the program attempted to use the specified DOS function call, and that call is not supported by the DOS extender. If this is not the expected behaviour please contact Blink, Inc technical support with your Blinker serial number and details of the problem.

1304: cannot use XMS host with current memory manager configuration

---

This message indicates that the DOS extender cannot switch to protected mode due to the configuration of 386 memory management software.

When a 386 memory manager is present, the DOS extender is not permitted to use its own mode switching routines to enter and leave protected mode. Instead, it must use either VCPI or DPMI mode switching services provided by the memory manager software.

This message indicates that a 386 memory manager (or similar) is running on the system, but there are no DPMI or VCPI services available to the extender (or the extender has been configured not to use those services through the link time commands, or through the BLINKER environment variable).

Solutions:

If possible, configure the memory manager to provide VCPI services (hint: enabling EMS memory will usually also enable VCPI).

If possible, configure the memory manager to provide DPMI services Ensure the extender's use of VCPI or DPMI has not been disabled by link time commands (eg: BLINKER HOST VCPI OFF), or by runtime switches contained in the BLINKER environment variable.

Very old memory managers may not be able to provide VCPI services such memory manager may need to be removed before the Blinker extender can operate.

1305: maximum selector limit exceeded

This message indicates that the maximum number of selectors for a program (approximately 8000) has been exceeded. This will typically be caused by an application program allocating a large number of small segments.

The solution is to modify the program to allocate larger segments and manage them by subdividing them inside the program, in the same ways as the C standard library malloc and free functions.

1306: invalid selector value

This message indicates that an invalid selector number was passed to an internal extender function, and should not occur in normal operation.

1307: unable to load segment

This message indicates that the DOS extender loader was unable to load a segment when a segment not present exception occurred. This is similar to being unable to load an overlay, and typically indicates that the .EXE file has become corrupt or has been overwritten. Please attempt to reinstall or recreate the .EXE file which gave the error.

1308: execution terminated by user

This message indicates that the user pressed the Ctrl-C or Ctrl-Break keys and the DOS extended program had not installed a Control Break handler, so execution was terminated.

1309: no more memory available

This message indicates that the temporary swap file created by the virtual memory system has filled the disk on which it was created, or that the virtual memory sytem limit of 64 MB has been reached.

Check that the program is not looping allocating memory, and if not then make more space available on the temporary drive or change the drive to which the virtual memory swaps.

1310: this program requires a 286 processor or above

This message indicates that the DOS extended program is not being run on an 80286 processor or above.

Protected mode is only supported on 286 processors and above.

1311: this program requires DOS 3.0 or later

This message indicates that the DOS extended program is being run under a version of DOS prior to 3.0, which is not supported.

1312: insufficient extended memory available to run

This message indicates that there is insufficient extended memory available to load and run the program. The amount of memory to run the program will typically depend on the size of the program, but a minimum may have been set using the BLINKER EXECUTABLE EXTENDED nnnn command at link time.

1313: exception error nn : <exception type>, code = nnnnh

This message indicates that a protection exception of the specified type has occured during the execution of the program. This is an application program error not a DOS extender error, and indicates that the application program has attempted an unauthorised operation.

The exception types and a brief description of their causes are listed below. The numeric code will often indicate the value causing the error, for example an invalid selector being loaded into a segment register or a pointer beyond the end of a segment.

This error message will be followed by a dump of all the processor registers and various other information. Analysis of this information is covered in more detail in Chapter 5, DOS Extended / Dual Mode Programs.

Exception 00: divide by zero attempted

This exception is generated by an attempt to divide a value by zero, and is normally handled by the compiler run time library when a high level language is being used.

Exception 05: bounds check failed

This exception is generated by a BOUND instruction when the value to be tested is less than the indicated lower bound or greater than the indicated upper bound.

Exception 06: undefined opcode

This exception is generated by an attempt to execute an opcode which is not defined for the processor the program is being run on.

Exception 07: no maths coprocessor available

This exception is generated by an attempt to execute a floating point instruction when the processor has no maths coprocessor available.

Exception 08: double exception fault

This exception is generated when multiple exceptions occur on one instruction, or an exception occurs in an exception handler.

Exception 0a: invalid task switch state segment

This exception is generated by an attempt to task switch to a segment with an invalid task state segment.

Exception 0c: stack overflow / underflow

This exception is generated by a limit violation in any operation which refers to the stack segment. This includes stack-oriented instructions such as PUSH and POP as well as other memory references which implicitly use the stack such as MOV AX,[BP].

Exception 0d: general protection fault

All protection violations which do not cause another exception cause a general protection exception. These include exceeding segment limits, attempting to load invalid selectors into segment registers, attempting to execute data and attempting to write to code.

Exception 0e: page fault

This exception is generated by an attempt to load a segment to an invalid memory address.

1314: unable to create file <filename>

This message indicates that the DOS extender virtual memory system was unable to create the specified temporary swap file.

The most likely causes of this error message are that there is already an existing file of the same name which has been assigned read only status, or no create access rights have been granted to the destination drive or directory, or that the destination disk is full.

1315: insufficient conventional memory to run program

This message indicates that the Dual mode program has attempted to load in real mode but has found that there is insufficient conventional (DOS) memory to do so.

Free up conventional memory by removing TSRs etc before running this program, or refer to the Blinker manual for details of how to overlay more code for a real mode program.

One possible cause of this error is that no overlays have been specified at link time, resulting in a large loadsize when the program is executed in real mode.

1316: VM swap file disk full or VM limit exceeded

This message indicates that the Blinker DOS extender virtual memory system is attempting to swap to disk and the destination is invalid or full, or the maximum swap file size of 64MB has been exceeded.

By default the swap file will be created in the directory specified by the TMP or TEMP environment variables, or the current directory if neither of these are set. Check that they point to a valid disk with sufficient available space for the DOS Extender's temporary VM swap file.

1317: extender heap full

This message indicates that the Blinker DOS extender heap has been completely used up, and should not occur in normal operation.

It usually indicates that a large number of small allocations of memory have been made directly via the extender API, which in turn is usually caused by repeated allocation of a temporary buffer without it being freed each time.

Check the program for endless recursion or failure to free a temporary memory allocation in a function which is called repeatedly.

1318: divide by zero attempted at (nn) ssss:oooo

This message indicates that the code in .EXE segment nn at address ssss:oooo executed an integer divide instruction with a divisor of zero and a divide by zero handler had not been installed.

[Next Entry>>]

Other common errors

DOS open error 4

This is a DOS error that indicates there are no more handles left to open another file. Increase the number of file handles in each of three places:

- The workstation - FILES=nn in CONFIG.SYS
- The network - (SHELL.CFG for Novell)
- The program - some C compilers limit the number of file handles in the start-up routine.

Remember that Blinker requires one file handle per overlay: one for an internal overlay or when using external overlays one per SECTION INTO command. Also, CA-Clipper 5.x requires a corresponding number of file handles for the dynamic paging system.

DOS open error 5

Particularly for DOS 4.0 and 5.0 where SHARE is automatically loaded, a fixed sized table is allocated that will contain the file name and full path for each open file. This error often indicates that there is no more room in this table to open another file. In this case, execute the DOS SHARE command with an enlarged space for the /f parameter.

Sharing Violation

Certain networks including 3COM and LANTASTIC encounter sharing violation errors with internal overlays when a second copy of the program is loaded. This appears to be due to the network requiring exclusive write access to the program file when it loads it.

To circumvent this problem either set the .EXE to read only access by using the Blinker command READONLY (or the DOS ATTRIB command) or specify an external overlay with the Blinker command SECTION INTO at the start of the BEGINAREA in the script file.

[^^Up^^]

[<<Previous Entry]

Runtime environment settings

Various aspects of Blinker linked programs may be overridden at program runtime through the use of BLINKER environment variable (the name of which may be changed using the BLINKER ENVIRONMENT NAME command at link time).

All BLINKER environment settings begin with a '/' character, and should be separated by spaces eg:

```
SET BLINKER=/HM+ /EL1000
```

Overlay Manager Settings

/CEnnn,nn (EMS overlay cache size)

This setting controls the size of the EMS overlay cache. See the BLINKER CACHE EMS command for more details.

/CXnnn,nn (XMS overlay cache size)

This setting controls the size of the XMS overlay cache. See the BLINKER CACHE XMS command for more details.

/DHn (debug handle)

This setting specifies the file handle to be used as the destination of debug output from BLINKER. See the BLINKER DEBUG HANDLE command for more details.

/DI3 (debug interrupt 3)

This setting (DEBUG INT 3) enables the creation of a debug interrupt 3 instruction just before the Blinker kernel jumps to the first line of the main program. This setting should only be used while debugging, and for more details see the file DEBUG.DOC

/OOnnn (overlay operating size)

This setting controls the size of the BLINKER overlay pool. See the BLINKER OVERLAY OPSIZE command for more details.

/OP+ and /OP- (overlay pageframe)

This setting enables and disables the use of the EMS pageframe for the overlay pool. See the BLINKER OVERLAY PAGEFRAME command for more details.

/OU+ and /OU- (overlay UMB)

This setting enables and disables the use of UMBs for the overlay pool. See the BLINKER OVERLAY UMB command for more details.

DOS Extender Settings

/DTnnn (dual mode threshold)

This setting allows the dual mode threshold specified at link time to be overridden. This can also be used during testing to force the program to run in real mode by setting /DT20000 (or some suitably large figure). See the BLINKER EXECUTABLE DUAL command for more details.

/EBnnn (extended memory block size)

This setting controls the size of the memory blocks allocated by the extender from the host. The default size is 256KB. See the BLINKER EXTMEM BLOCK command for more details.

/ED+ and /ED- (extender debugging)

This setting enables and disables certain debugging messages provided by the extender, such as checking for aliasing of segments which are not locked or not present, and freeing of selectors using a free memory function call. It is advisable to test your programs running under a VCPI or XMS host (ie not DPMI) with this option enabled to check for these circumstances. The default is disabled.

/EInnn[,nnn] (extender IPX buffers)

This setting specifies the number of IPX ecb buffers, and optionally the number of IPX send buffers, allocated by the extender's IPX module. See the BLINKER EXECUTABLE IPX command for more details.

/EK+ and /EK- (extender keyboard handling)

This setting enables and disables special handling of certain keyboard BIOS calls by the extender to prevent needless mode switching. When enabled, this option can increase the performance of serial communications routines. When disabled, this option may improve compatability with old TSRs which manipulate the keyboard. The default is disabled.

/ELnnnn (extended memory limit)

This setting specifies the total amount of host memory that the extender will allocate before the extender's VM system begins to swap to disk in order to satisfy new allocation requests. This setting can be used to limit the amount of host memory used, or to simulate running in lower memory conditions. See the BLINKER EXTMEM LIMIT command for more details.

/ETnnnn (extended memory threshold)

This setting allows the extended mode threshold specified at link time to be overridden. See the BLINKER EXECUTABLE EXTENDED command for more details.

/HD+ and /HD- (host DPMI)

This setting enables and disables the use of a DPMI host. See the BLINKER HOST DPMI command for more details.

/HM+ and /HM- (host messages)

This setting enables and disables the displaying of status messages at program start up. See the BLINKER HOST QDPMI command documented above for more details. The default is disabled.

/HP+ and /HP- (host passup)

This setting enables or disables the Blinker extender's automatic pass up of IRQ interrupts from real mode to protected mode. This setting should only be

necessary for debugging purposes when testing interrupt handlers. See also the BLINKER HOST PASSUP command. The default is enabled.

/HQ+ and /HQ- (host QDPMI)

This setting enables and disables the use of the Quarterdeck QDPMI host. See the BLINKER HOST QDPMI command for more details.

/HV+ and /HV- (host VCPI)

This setting enables and disables the use of a VCPI host. See the BLINKER HOST VCPI command for more details.

/HX+ and /HX- (host XMS)

This setting enables and disables the use of an XMS host. See the BLINKER HOST XMS command for more details.

/LRnnn (lowmem reserve)

This setting specifies the amount of conventional memory reserved for program use, preventing Blinker's DOS extender from using it for other purposes. See the BLINKER LOWMEM RESERVE command for more details.

/WS+ and /WS- (Windows Soft-Ice)

This setting enables or disables the extender's check for the presence of the Soft-Ice/W debugger in protected mode running in a Windows enhanced mode DOS box. The default is enabled.

[Next Entry>>]

Swap / Run package settings

/SD<drive:\path> (swap path)

This setting specifies the path where the temporary file created by the swapper will be created. See the function SWPRUNCMD() for more details.

/SE+ and /SE- (use of EMS memory)

This setting enables and disables the use of EMS memory by the swap package. See the function SWPUSEEMS() for more details.

/SU+ and /SU- (use of upper memory blocks)

This setting enables and disables the use of upper memory blocks by the swap package. See the function SWPUSEUMB() for more details.

/SX+ and /SX- (use of XMS memory)

This setting enables and disables the use of XMS memory by the swap package. See the function SWPUSEXMS() for more details.

[^^Up^^]

[<<Previous Entry]

Blinker environment setting /DI3

Setting /DI3 (DEBUG INT 3) in the BLINKER environment variable causes a debug interrupt 3 to be created just before the far jump to the first byte of the actual program. This is after both the extender and the Blinker Kernel have loaded and initialised. This instruction can be stepped into in the normal way to take you to the first byte of the actual program.

In high level language programs this will be the library start up code, so to skip over the library start up to the start of your program (function MAIN) type 'G MAIN' or its equivalent.

Microsoft CodeView

Overview

Note: The Microsoft CodeView debugger CANNOT be used to debug DOS Extended programs - instead use a debugger such as Soft-Ice/W or Periscope/32 for Windows.

Blinker supports CodeView debugging of REAL mode programs with compilers that produce CodeView information levels 0, 1, 2 and 5. This encompasses all MS compilers through C/C++ version 7.x and Visual C++ (C8), Visual BASIC for DOS 1.0, BASIC PDS 7.1, Pascal 4.0, Fortran 5.0 and MASM 6.x. At the time of writing, other manufacturer's compilers that produce CodeView information comply with this requirement. To enable Blinker's processing of CodeView information, add the following command to your link script:

```
CODEVIEW
```

CodeView Versions

CodeView version numbers in this discussion relate to the version number of the information, not to any particular revision of the CodeView debugger software.

The CodeView information that Blinker produces by default conforms to the lowest level of support, that is CodeView version 0. This level of information is supported by all versions of CodeView and any utility that makes use of CodeView information. However, later versions of the CodeView debugger may support additional features that are not suppported by information level 0. In this case you will need to instruct Blinker to produce information to a higher level than the default. The CVVERSION command is used for this purpose:

```
CVVERSION 5    # Provide level 5 Codeview support as
               # required by Microsoft Visual C++
```

Valid CodeView versions are 0, 1, 2 and 5. Do not use a CodeView version number higher than that which your debugger supports.

CodeView and Overlays

Note: you must always use external overlay(s) when debugging overlaid programs under CodeView.

The CodeView information standard prior to version 5 makes no provision for the concept of a dynamic overlay linker like Blinker, the reason being that Microsoft overlay linkers could create only statically overlaid programs. Version 5 is supported by the CodeView that comes with Microsoft Visual C++, and by most compatible compilers, so users of those compilers need read no further.

For older versions of CodeView the primary difference is that with a statically overlaid program, the address at which an overlay will execute can be determined at program link time, as it is within the executable image, whereas a dynamically linked program executes overlays at an address the can only be determined at program run time.

In order for Blinker to provide compatibility with the CodeView standard, and still retain the advantages of dynamic overlaying, Blinker creates a static overlay area, the size of which is determined by the largest overlaid object module, within the executable image. At program execution time, the overlay manager will identify those modules that are within a 'CodeView area' and copy them into the static area for execution. The creation of the static overlay area within the program image will increase the loadsize of the program by the size of the largest overlaid module for which codeview information is active.

In order to allow you to single step and trace into overlaid modules, Blinker will create a source file with the same name as the executable, but with the extension .VEC. This file should be made available to CodeView at program execution time, as if it were one of your own source files. When tracing your program, follow the directions given in this source file as to when to step over and when to step into the overlay vectors and overlay manager.

There is a penalty in execution speed while debugging overlaid code under the older versions of CodeView, which is primarily caused by the overlay manager having to copy modules into the static overlay area. In effect this causes a worst case overlay situation whereby modules are being constantly swapped in and out of a single memory area, the impact of which will vary depending on the program (and modules) being debugged. There are two things that can be done to help alleviate this problem:

- Limit the amount of code that is to be debugged by using CodeView areas
- Increase the size of the overlay cache to speed up overlay operations

BLINKER ENVIRONMENT CLIPPER   Change the name of the CA-Clipper environment variable

---

Purpose : **Change the name of the CA-Clipper environment variable**

Syntax : **BLINKER ENVIRONMENT CLIPPER <cname>**

Example : **# Change the name of CLIPPER variable to Newclip**
BLINKER ENVIRONMENT CLIPPER Newclip

This command allows the developer to change the name of the CLIPPER environment variable from the default setting of 'CLIPPER' to any string of up to 16 characters, and thus avoid the remote site settings for the CLIPPER environment variable.

Used in conjunction with the BLINKER ENVIRONMENT OVERRIDE command, this allows the settings burnt in with the BLINKER EXECUTABLE CLIPPER command to be overwritten at runtime by this newly created variable.

The choice of a unique name avoids any conflict with previous settings for the CLIPPER environment variable.

For example, if a script file contains the following commands:

```
BLINKER EXECUTABLE CLIPPER F31;X100
BLINKER ENVIRONMENT OVERRIDE
BLINKER ENVIRONMENT CLIPPER ACCSYS
```

Even though the following statement may already have been set from DOS:

```
SET CLIPPER=F21;
```

this CLIPPER environment variable will be ignored by the application, and the BLINKER EXECUTABLE CLIPPER setting can be overridden by the following DOS command:

```
SET ACCSYS=F99;
```

See Also:

NuMega Soft-Ice/W

Overview

Blinker 3.2 includes full support for the Soft-Ice/W debugger for real, protected, dual and Windows programs on a SINGLE MACHINE. Soft-Ice/W actually loads and runs Windows and sits at the lowest system level, so it can debug anything at the source level, including VxDs and DOS protected mode programs.

To use Soft-Ice/W with DOS real mode or protected mode programs simply open a DOS box, link for CODEVIEW with CVV 5, run WLDR xxxx.SYM (A SYM file is not actually needed) to load the symbols, source files etc and then run the program xxxx.EXE directly or WLDR xxxx.EXE. Do not use WLDR xxxx with no extension as there seems to be a problem loading DOS extended programs at the same time as their symbols.

To enable the interception of the INT 3 debugging interrupt by Soft-Ice/W use the command 'I3HERE ON' at the Soft-Ice/W command line (pop up Soft-Ice/W using Ctrl D), or by including it in the INIT line in the Soft-Ice/W initialisation file WINICE.DAT.

Soft-Ice/W will then pop up when it encounters an INT 3 in the code, when a Control D is pressed, or it can be forced to pop up at the very start of the program by setting the value /DI3 in the BLINKER environment variable as described in the previous section. Soft-Ice/W will then pop up on a far JMP to the very first line of the actual program, which can be stepped into with F8. As noted above, for high level languages the command 'G MAIN' will take you to start of the main program.

Periscope and Periscope/32

Overview

Blinker 3.2 includes full support for the Periscope debugger for real mode programs and the Periscope/32 debugger for real, protected, dual and Windows programs using two machines. Periscope/32 requires one machine to run the program being debugged and a second machine, on which the source code is stored, to run the debugger itself.

To use Soft-Ice/W with DOS real mode or protected mode programs simply link for CODEVIEW with CVV 2 or 5, run TS /e xxxx and then run the program xxxx.EXE directly or RUN xxxx.EXE.

Reducing debugging information using CodeView areas

CodeView areas

When the CODEVIEW command is encountered in the link script, Blinker will assume that you wish to debug all modules in the program. In order to limit the number of modules for which this debugging information is processed and stored in the .EXE file, you may wish to create CodeView areas that contain only the program modules you wish to debug. This is accomplished through the use of the CVON and CVOFF commands, which specify areas to be debugged.

For example:

```
CODEVIEW                    # Enable CodeView
FILE start                  # Can be debugged
BEGINAREA
   SECTION INTO START       # External overlay required for
                            # CodeView support
   CVON
      FILE one              # Can be debugged
      FILE two              # Can be debugged
      FILE three            # Can be debugged
   CVOFF
   FILE four                # Cannot be debugged
   FILE five                # Cannot be debugged
   FILE six                 # Cannot be debugged
   FILE seven               # Cannot be debugged
   FILE eight               # Cannot be debugged
   CVON
      FILE nine             # Can be debugged
   CVOFF
   LIB inhouse              # Cannot be debugged
ENDAREA
```

Problems when debugging

Blinker error 1203

Some debuggers require exclusive access to the executable file during debugging, which may cause problems with overlaid programs, the usual symptom of which is a Blinker error 1203. To alleviate this problem, use the Blinker READONLY script command to create the executable with a read only attribute, or use the DOS ATTRIB command to set the read only attribute following the link.

When debugging overlaid programs, ensure you are using the current version of the vector source file (.VEC), otherwise things may become confusing!

Overlays and debuggers other than CodeView

When the overlay manager loads or unloads an overlay, it informs CodeView that a section change has occurred. If your debugger, or other utility is not fully compatible with CodeView, you may not be able to use it to debug overlaid programs, although you should still be able to debug modules in the root.

New since the manual

2. Instructions for specific compilers

3. DOS extender notes

4. New commands, functions and error messages

[<<Previous Entry]

1. General Notes

Blinker now fully supports the creation of 16 bit OS/2 .EXE and .DLL files. To create OS/2 .EXEs and .DLLs simply create a link file as documented in the manual and use the module definition file(s) from your current OS/2 linker.

Programs linked with the Blinker 3.x DOS extender require a minimum of HIMEM.SYS to be installed on the destination machine in order to run in protected mode. If there is no memory manager at all present then dual mode programs will revert to running in real mode and extended mode programs will terminate with the following message :

> This is a DOS extended program which requires a minimum of a 286 processor and at least nnnn KB of available DPMI, VCPI or XMS memory to run. Please ensure that your system is configured with a memory manager which provides these resources.

All other memory managers such as QEMM, 386MAX , NetRoom etc are fully supported and are used to their full potential.

Blinker searches for FILEs in the OBJ path AND THEN the LIB path before failing with a file not found error.

2.1. Borland C++ 4.x

SEARCHing of the Borland C++ libraries or other libraries created with TLIB 4.x is no longer necessary, nor is the link command NOTABLEOFCONTENTS. Blinker 3.2 now fully supports these types of libraries.

The OBJ subdirectory contains the .OBJ files FP7BCPxx.OBJ which must be linked to any protected mode programs which use LIB FP87 instead of LIB EMU to force use of the maths coprocessor for floating point operations. These version specific files replace the generic BLXBCP87.OBJ from previous versions of Blinker.

2.2. CA-Clipper 5.x

The ONLY release versions of CA-Clipper 5.x supported by Blinker 3.2 are 5.01a, 5.2d/e and 5.3. 5.2c is no longer supported.

CA-Clipper 5.3 Workbench link templates for real mode, dual mode and extended mode programs are provided in the CLP subdirectory in files BLIRMD.TPL, BLIDUA.TPL and BLIEXT.TPL respectively. Note that incremental linking will default to ON when using these templates.

2.3. Microsoft C/C++/FORTRAN

The Microsoft C/C++/FORTRAN graphics libraries for version 5.1 and later are fully supported in protected mode.

2.4. Microsoft PASCAL

DOS extended programs can be created with Microsoft PASCAL 4.0 using the protected mode library BLXMSC51.LIB (Microsoft C 5.1).

2.5. Stony Brook Compilers

The compiler run time libraries, and any libraries produced by the Stony Brook librarian must be SEARCHed. Use of the LIB command on these libraries will result in an internal Blinker error 1191: invalid library number. Alternatively, the libraries may be rebuilt using the Microsoft librarian (LIB.EXE).

When using overlays, the startup module must be in the root.

Modula 2

When using overlays, compile using the LARGE model:

```
m2 myprog.mod /CODE:L
```

Pascal+

When using overlays, compile using the LARGE model, and place constants in a separate segment:

```
ppc myprog.pas //CODE:L //CONSTANT:L
```

2.6. Watcom C/C++ 10.x

Watcom C++ 10.x (16 bit) is fully supported in real mode, protected mode and dual mode using the library BLXWCC10.LIB.

Note that the file FP7WCC10.OBJ MUST be explicitly linked in addition to BLXWCC10.LIB when NO floating point emulation is required in a protected mode program, ie when compiling with options -fpc, -fpi87, -fp2 or -fpr.

3.1. Memory requirements and memory managers

Although the Blinker 3.x DOS extender defaults to looking for a DPMI host first for maximum compatibility, a VCPI or XMS host may provide more memory to a DOS extended application in tight memory situations. As a result, DOS extended programs may run better on smaller machines when only HIMEM.SYS is installed.

In particular QDPMI allocates around 70kb of conventional memory when a program starts up, and this cannot be swapped out by the SWPRUNCMD() functions. As a result, QDPMI support is automatically disabled when the swapper is linked into a program. This default setting can be overidden with the BLINKER HOST QDPMI command or the /HQ environment setting.

Note: When creating a menu program which then swaps out to a DOS extended program it is more memory efficient to make the menu program a real mode program and use the SWPRUNCMD() than to also make the menu program a DOS extended program.

3.2. Add on / 3rd party libraries

Please note that many add on libraries written in C or Assembler will require updates to allow them to be used in protected mode and dual mode programs.

When a library is overlaid in a Dual mode program, a special link script or a root portion of the library may be required, in the same way as with a normal overlaid program. Please ensure that you are using the correct link script or library file.

The file 3RDPARTY.DOC on the distribution disk contains the latest information we have available. The information was supplied by the vendors concerned, so is provided on an as is basis and Blink, Inc makes no guarantee that the information is correct.

Note: All the overlay manager commands and functions are still applicable to a Dual mode program when it is running in real mode, so be sure to set up an overlay cache, enable OVERLAY PAGEFRAME if possible and set a sufficient overlay area size etc.

3.3. General Protection Exceptions / Faults (GPEs / GPFs)

General Protection Exceptions can occur when the program code currently executing attempts to perform an invalid operation, access an invalid segment or access beyond the end of a valid segment. These errors ARE NOT extender or linker errors, they are errors in the protected mode program.

GPF dumps are written to the standard output file so can be redirected to a file by redirecting a program's output on the command line. E.g.

```
TEST > TEST.GPF
```

The SEGMENTS module definition command (see the Windows programming chapter in the manual) can also be used with the DOS extender for interrupt handlers etc to force individual segments to be kept separate and loaded in conventional memory by specifying them as FIXED.

E.g.

```
BLIINTHAN SEGMENT DWORD PUBLIC 'BLIINTHAN'

BLIINTHAN ENDS
```

To force it to be fixed :

```
DEFBEGIN

SEGMENTS 'BLIINTHAN' CLASS 'BLIINTHAN' FIXED

DEFEND
```

This can also be very useful to isolate the source of a GPF once you have found the segment it is in. Just do the above for the segment where the problem is occurring and it will be placed in a segment on its own, so the offset displayed by the dump will tie in with the offsets from your assembly source and listing files.

When a Dual mode program has a GPE it may occur in an overlaid segment in which case the CS value will be relatively high and no segment number will be displayed. In this case relink the program as an Extended mode program to remove overlays and put the segments in the map where the dump function can identify them. Alternatively you can take the code segment limit from the dump and refer to the table of overlays in the MAP file to locate the most likely overlay.

3.4. DPMI API functions

If you would like to program directly to DPMI, the following DPMI calls are supported by the Blinker 3.x DOS Extender, whatever the host memory manager is :

```
0000h Allocate LDT descriptors
0001h Free LDT descriptor
0002h Real segment to selector
0003h Get selector increment
0004h Lock selector
0005h Unlock selector
0006h Get segment base address
0007h Set segment base address
0008h Set segment limit
0009h Set descriptor access rights
000Ah Create data alias for Code selector
000Bh Get descriptor
000Ch Set descriptor
0100h Allocate DOS memory block     (Note: Blocksize limited to 64 Kb)
0101h Free DOS memory block
0102h Resize DOS memory block       (Note: Blocksize limited to 64 Kb)
0200h Get real mode interrupt vector
0201h Set real mode interrupt vector
0202h Get processor exception vector
0203h Set processor exception vector
0204h Get prot-mode interrupt vector
0205h Set prot-mode interrupt vector
0300h Simulate real mode interrupt
0301h Call real mode procedure with RETF frame
0303h Allocate real mode call back address
0304h Free real mode call back address
0400h Get version
0500h Get free memory information
0600h Does nothing, returns with carry clear
0601h Does nothing, returns with carry clear
0602h Does nothing, returns with carry clear
0603h Does nothing, returns with carry clear
0604h Does nothing, returns with carry clear
0A00h Get extended API address

Call in protected mode with DS:SI -> 'BLINKINC',0

Returns with carry Clear,

ES:DI -> API entry point
```

4.1 New commands

BLINKER HOST PASSUP [ON | OFF]

This command enables or disables the Blinker extender's automatic pass up of IRQ interrupts from real mode to protected mode. This command should only be necessary for debugging purposes when testing interrupt handlers, and is enabled by default. See also the new /HP+ environment setting.

4.2 New environment settings

/DI3 (DEBUG INT 3)

This setting enables the creation of a debug interrupt 3 instruction just before the Blinker kernel jumps to the first line of the main program. This setting should only be used while debugging, and for more details see the file DEBUG.DOC

/HP+ and /HP- (HOST PASSUP)

This setting enables or disables the Blinker extender's automatic pass up of IRQ interrupts from real mode to protected mode. This setting should only be necessary for debugging purposes when testing interrupt handlers, and is enabled by default. See also the new BLINKER HOST PASSUP command.

/WS+ and /WS- (WINDOWS SOFTICE)

This setting enables or disables checking for the presence of the Soft-Ice/W debugger in protected mode running in a Windows enhanced mode DOS box. This setting is enabled by default.

[Next Entry>>]

4.3 New error messages

1016 : ignoring duplicate import for symbol <symbol>

This message indicates that a duplicate import definition for the specified symbol has been encountered but ignored.

1017 : ignoring duplicate definition for segment <segment>

This message indicates that a duplicate segment definition for the specified segment name has been encountered but ignored.

1018 : unable to export symbol <symbol>

This message indicates that an export definition for the specified symbol has been encountered but the symbol does not exist in the program so the export definition was ignored.

1150: <filename> is a demo so cannot be linked with this Blinker

This message indicates that the specified library is a special demonstration library which can only be linked with a demonstration version of Blinker. The demonstration version of Blinker should have been supplied along with the demonstration library.

1151: increase PACKDATA parameter to reduce SYMBOLs segments

This message indicates that there were too many SYMBOLs segments in an incrementally linked CA-Clipper program, so it is necessary to increase (or add) the parameter to the PACKDATA link script file command.

1152: too many exported name symbols

This message indicates that the .EXE or .DLL file being created contains too many exported name symbols to be placed in the .EXE header.

Either remove some EXPORT commands from the module definition file or remove one or more import libraries from the link script file.

1153: 3.2 requires a more recent version of BLXxxxnnn.LIB

This message indicates that the indicated library is from an older version of Blinker.

Check that the link script file and the LIB environment path point to the latest version of the indicated library.

1317: extender heap full

This message indicates that the Blinker DOS extender heap has been completely used up, and should not occur in normal operation.

It usually indicates that a large number of small allocations of memory have been made directly via the extender API, which in turn is usually caused by repeated allocation of a temporary buffer without it being freed each time.

Check the program for endless recursion or failure to free a temporary memory allocation in a function which is called repeatedly.

1318: divide by zero attempted at (nn) ssss:oooo

This message indicates that the code in .EXE segment nn at address ssss:oooo executed an integer divide instruction with a divisor of zero and a divide by zero handler had not been installed.

Overview

Protected mode

Protected mode refers to the enhanced mode of the 80286 and above. These processors were designed with backwards compatibility as being of primary importance, but with the need to extend the address space beyond 1 Mb.

When these processors are powered up they start running in 8086 mode, known as real mode, and in the absence of any special software will continue to run as an 8086, only with better performance due to their increased clock speeds and reduced instruction execution times.

Specific software instructions may be used to switch the processor into one of its enhanced modes at any time, at which point the processor starts to function as a more advanced processor with a number of extra capabilities, including the ability to access more than 1 Mb of memory. The processor may be switched in and out of protected mode at any time if required, although the 80286 requires a significant amount of time (milliseconds) each time a switch to protected mode is requested.

When running in 80286 protected mode, rather than interpreting the segment as an absolute memory address the processor interprets it as an index into a look up table which contains 24-bit physical base addresses for all the memory segments in the system. Using this 24 bit base address and a 16 bit offset allows the processor to address 16 Mb of physical memory. The memory is still segmented, however, with the largest segment still being 64 Kb.

DOS Extenders

The 80286 processor and above will run in real mode until otherwise instructed, and DOS itself is written in such a way that it can only run in real mode. The idea behind a DOS extender is to run the actual program in protected mode, giving it access to all the extra memory and features available, and then temporarily switch back to real mode whenever a DOS or BIOS service is required which cannot be fielded in protected mode.

A DOS extender will typically have a real mode code portion loaded in DOS conventional memory and a protected mode code portion loaded in extended memory. The real mode portion will load and execute the user program, and will receive control any time a request for, or access to, a system resource is made by the program. The protected mode portion will usually contain replacement code for some of the services offered by DOS or the BIOS, so in many cases it will perform the function and then return to the user program without having to leave protected mode.

If the request will require DOS or the BIOS to be activated, however, the DOS extender will save the current protected mode status of the processor, switch back to real mode and then invoke its real mode portion. This portion may then field the request itself or pass it on to DOS or BIOS in the normal way. Once the request has been processed, control returns to the real mode portion which will return to protected mode and the protected mode portion. This can now restore the protected mode status and continue execution of the user program.

DOS extenders are able to use the specific features of the 80286 or the 80386, and obviously require a minimum of the processor for which they have been written. In all cases, because the 80x86 processors are backwards compatible, the DOS extender will also run on any processor more advanced than the one for which it is written.

80286 DOS extenders run in 16 bit protected mode, which provides access to up to 16 Mb of memory in 64 Kb segments. Because it is still managed in 64 Kb segments, most high level language code will run in 16 bit protected mode

with only minor modifications and incompatibilities. Typically a recompile and relink with some replacement .OBJ and .LIB files provided as part of the DOS extender will allow a program to be up and running in 16 bit protected mode, at which point the programmer can start to iron out the incompatibilities.

80386 DOS extenders run in 32 bit protected mode, which provides up to 4 Gb of memory in one or more segments, so most programs will treat the memory as a linear address space contained in one segment. Because the address space is linear and segments are not limited in size to 64 Kb, more major incompatibilities in existing programs will arise. In the case of high level languages, 32 bit versions of the compiler are usually necessary in order to make use of 32 bit protected mode, and these compilers will automatically prohibit most of the constructs which are not allowed when executing in protected mode.

Windows 3.x enhanced mode is actually this 32 bit protected mode. We will make no further mention of 32 bit protected mode as the DOS extender provided with Blinker is a 286 DOS extender which will run on any processor from an 80286 processor with 2MB of memory up to the largest Pentium.

Because of the potential problems in a DOS environment of multiple programs wishing to switch back and forth into protected mode, particularly between EMS emulators and DOS extenders, two major programming interfaces were proposed to manage protected mode switching. The first of these was the Virtual Control Program Interface (VCPI) which is specific to 80386 and 80486 processors, and the second and more general is DOS Protected Mode Interface (DPMI), which applies to the 80286 and above. Please refer to Appendix C - Further reading - for more details on these specifications and how to program to them.

Performance implications

Programs running under a 286 DOS extender have transparent access to up to 16MB of directly addressable memory. Memory intensive programs, particularly those using large numbers of data structures e.g. arrays, will therefore run significantly faster in protected mode. Also, since there is usually no need for overlaying of code, execution of the code itself is faster because pauses to load code from disk are rare.

On the other hand, programs running under a DOS extender which heavily access DOS, the BIOS or other external resources will often be continually switching between real and protected mode, so may run slower than in real mode. This is extremely program dependent. The overhead of switching between real and protected mode varies significantly between different machines, and reduces drastically when moving from a 286 to a 386 and 486 processor.

Language compiler memory managers sometimes perform inefficiently when given megabytes of data, since they were only designed to manage a few hundred Kb of data. This is another area where performance may be adversely affected. For this reason Blinker provides replacement memory management functions for some of the more popular compilers.

BLINKER ENVIRONMENT NAME   **Specify the name of the environment variable for Blinker run time options**

Purpose : **Specify the name of the environment variable for Blinker run time options**

Syntax   : **BLINKER ENVIRONMENT NAME <cname>**

Default : **BLINKER**

Example : **# Rename the Blinker variable**
BLINKER ENVIRONMENT NAME Testapp

and use the following command at the DOS prompt:

SET **TESTAPP=/O030**

to alter the OPSIZE at program runtime without having to relink.

This command is used to change the name of the BLINKER environment variable.

At program runtime the Blinker overlay manager checks the DOS environment for the BLINKER environment variable, and if this is present then the variable is examined for certain runtime parameters.

Following linking with the above command, the overlay manager will check the environment for the TESTAPP variable. For a listing of the BLINKER environment variable's valid parameters, please see Chapter 2.

See Also:

Coding for protected mode

Converting your code

The major point on converting your code is that the higher level the language, the easier it is to convert your code. High level languages such as CA-Clipper require no code changes whatsoever as the code actually executing is contained in the CLIPPER.LIB file itself. Medium level languages such as C, PASCAL, FORTRAN etc in their pure form also require no code changes. Almost the only time changes are needed is when the programmer has hard coded direct access to system resources, such as using an absolute pointer to a video buffer.

This type of code needs to be adapted slightly to store and use the pointer in a variable which is set up at the beginning of the program. For example, in the case of video writes, a system call would be made at start up to set up a pointer to the video buffer. This would either be an absolute pointer if the program is in real mode, or a selector if the program is in protected mode. From then on, all video memory accesses are made using the same pointer, regardless of the processor mode, so there is no further overhead in code or execution time.

Assembly code is the most likely to require conversion, as it provides direct access to system resources which may not be available in protected mode. The code requiring conversion is similar in nature to that previously described, it is simply more likely to be prevalent in assembly code.

A final construct requiring conversion relates to segment addresses. In protected mode segments are not allocated sequentially and are not contiguous, thus comparison of segment addresses is invalid. For example, sequential allocation of segments from the DOS extender will at best produce selectors separated by an increment of 8, and at worst produce them in a seemingly random sequence. This is because the selectors are simply pointers into an array known as a descriptor table, and the low three bits of the selector are used to store control information. For more details see the DosGetHugeShift() function in Chapter 10.

To put this conversion process in perspective, in case it sounds an enormous task, we recently converted Blinker to run in dual mode. Blinker currently consists of approximately 60,000 lines (4MB) of pure assembly code, which is the worst kind of code to convert. The whole conversion to run in dual mode, ie normal DOS mode OR protected mode under our own extender, took one programmer less than a day. The code is now totally compatible with both modes with no conditionally compiled code, and the extra overhead is less than 100 lines called at various initialisation points at the beginning of the program.

Debugging under protected mode makes the detection of code to convert relatively simple, just run the code and see where it stops with a protection exception. Typically an error message will be displayed indicating the location and type of the error, and the program will return to DOS. When running under Windows 3.x or OS/2 2.x this can be done many times without requiring a reboot of the machine.

Special considerations for protected mode code

Incompatibilities with protected mode usually arise from certain restrictions which were designed into the processor to provide better protection and isolation between multiple programs running on the same processor. High level language compilers can often detect code which will break these restrictions, while other disallowed constructs will only become apparent when the code is actually executed at run time. Restrictions on coding techniques include :

- Attempts to access addresses outside (i.e. past the end of) valid segments will cause a protection exception.
- Attempts to directly modify code during execution, or to directly execute data, will cause a protection exception.
- Using segment registers for any other purpose than as segment registers. Some code uses the segment registers as a temporary store rather than writing it to memory. Whenever the value of a segment register is modified the value must be a valid selector or a protection exception will occur.
- Performing real mode address arithmetic e.g. pointer normalisation, since segments are no longer contiguous or even adjacent. This usually occurs when managing data items larger than 64kb, and must be re-programmed.
- All pointers to data buffers which will be passed to real mode code, such as when issuing a software interrupt or communicating with a TSR, must be valid real mode addresses, so the buffers must have been allocated from real memory below 1 MB. The only exceptions to this are the DOS INT 21 function calls which are handled transparently.

It should be emphasised again that occurrences of these coding constructs are definitely the exception rather than the rule, and that the higher level the language, the less likely one of these constructs is to occur.

Dual mode

Overview

The ability to create dual mode programs simply by adding a single command to the link script file is a major breakthrough. Dual mode programs detect at startup whether there are enough system resources to run in protected mode. If there are insufficient resources then the same program will run as a normal dynamically overlaid program.

The .EXE still only consists of one program capable of operating in either mode, it is not two separate copies joined together in some way. There is negligible increase in .EXE size over a single mode program, and a similarly negligible overhead in execution speed, since the whole program is loaded into memory when running in protected mode.

For this dual mode to work and be effective, all code must be written to execute in both real and protected mode. High level languages, such as CA-Clipper, require no code changes whatsoever to run in protected mode, and changes for C, PASCAL and FORTRAN are the exception rather than the rule. This requirement should only effect high level C and ASM programmers. In the worst case the code should be able to test for the processor mode each time it is executed and then jump to the appropriate code.

We suggest that C and ASM programmers DO NOT create a protected mode library separate from the real mode library, since both types of code will have to be linked in at link time to create a dual mode program. The benefits of writing dual mode code and creating a dual mode library include :

- Only one copy of the source code and one library to maintain.
- Less disk space requirement for distribution.
- Less technical support with people linking the wrong library.

Coding for dual mode

This section details the suggested way to adapt code to run in both real and protected mode, rather than coding specifically for one or the other.

When linking DOS extended or dual mode programs it is always necessary to include an additional Blinker library for the appropriate compiler. This library contains replacement .OBJs for any compiler library .OBJs which are not protected mode compatible.

The Blinker libraries all have names of the form BLXcccnn, where ccc indicates the vendor and compiler, and nn indicates the compiler version number. They are all compiled as large model and so should only be used with large model programs.

The appropriate compiler library must always be SEARCHed just before the standard compiler library or the script file containing the compiler library.

For example, with Microsoft Visual C++:

```
SEARCH BLXMVC15      # Blinker library
LIB LLIBCE           # MSVC library
```

or with CA-Clipper 5.2:

```
SEARCH BLXCLP52      # Blinker library
LIB CLIPPER        # CA-Clipper library
```

or:

```
SEARCH BLXCLP52      # Blinker library
@CL520MID            # Script containing CA-Clipper library
```

If the compiler library is not explicitly mentioned then place the SEARCH of the Blinker library as the last library in the link file.

When using third-party libraries please ensure that they are protected mode or dual mode compatible before attempting to use them in a DOS extended program. Blinkinc cannot guarantee that any particular library is either overlayable or protected mode compatible or both. Please refer to the file 3RDPARTY.DOC for up-to-date details on compatibility issues which we are aware of. The third party libraries should typically be placed before the SEARCH of the Blinker library unless otherwise documented in the third party product.

As stated before, most code will actually run in both modes without modification. Most of the remaining code will run in either mode after being modified to use a variable set up with the appropriate value at start up. For example, in the case of directly accessing the video buffer, in C we would write:

```
#include <blx286.h>
BYTE ProcMode;
static SEL Selector;
static unsigned char far *VideoBuffer;
...
if (DosGetMachineMode(&ProcMode) == 0)
   switch (ProcMode)
      {
      case 0:
         printf ("\nReal mode - absolute pointers ok\n");
         VideoBuffer = MK_FP(0xb800,0);
         break;
      case 1:
         printf ("\n286 Protected mode - plenty memory\n");
         if (DosMapRealSeg (0xb800, 4000l, &Selector))
            {
            printf ("\nTrouble\n");
            exit (1);
            }
         VideoBuffer = MK_FP(Selector,0);
         break;
      default:
         printf ("\nOther mode - interesting\n");
         exit(1);
   }
```

Then:

```
*VideoBuffer[0] = `*';     /* Display a `*' in top left */
                              /* corner of screen */
```

As can be seen, there is no extra overhead in code or execution speed once the initialisation has been done.

In the exceptional case where the code is totally different for the two modes, one of two approaches can be used. The first is to use a simple `if' or `switch' statement on a global variable, as is used in the example above for the initialisation code. If the choice has to be made throughout your code the second method of using a function pointer is more appropriate.

```
#include <blx286.h>

int (*DoCode) (int);        /* Variable for function pointer*/
int ProtCode (int);         /* Protected mode code function */
int RealCode (int);         /* Real mode code function */
...
if (ProcMode)               /* If protected mode */
```

```
      DoCode = ProtCode;
   else                        /* Real mode */
      DoCode = RealCode;

   int ProtCode (int abc)
      {
      printf ("ProtCode %d\n",abc);
      return (abc * 2);
      }
```

Then:

```
   DoCode(1234);               /* Execute the appropriate code */
```

It should be noted that in the unusual cases where an overhead is necessary, it is absolutely minimal, less than a millionth of a second in C even on a 286. In the case of a higher level language such as CA-Clipper, which executes hundreds of these tests for each `if' clause in the CA-Clipper source, the relative overhead is so small as to be totally irrelevant.

There is a common misconception that this type of construct will slow down a program written in a high level language program such as CA-Clipper. As anyone who has traced these things under an assembly language debugger will know, even passing a simple logical or integer variable from CA-Clipper to C or ASM requires hundreds of CPU instructions, including a large number of test and jump instructions like the one above.

The only time that the overhead is even vaguely an issue is when the code is executed hundreds of thousands of times without interacting with, i.e. calling or returning to, the high level language.

Low level programming

As an alternative to using the API functions described in Chapter 10, a subset of the DPMI API is also provided by the Blinker DOS extender if a DPMI host is not available. This means that programs can be written to directly access the DPMI functions to allocate and manage segments. The benefit is that the code is then not specific to a particular extender.

The down side is that the functions are typically at a lower level than those provided by the extender API functions, so more work may be required on the part of the programmer.

The Extender API

Naming conventions

All the extender API functions are prefixed with the letters `Dos' to indicate that they are part of the DOS extender API and are covered in more detail in the alphabetical function reference in Chapter 10. All these functions are intended to be called from languages which can use the Pascal calling conventions, precluding certain high level languages such as CA-Clipper.

Blinker also provides a generic get status function BLIMGRSTS() which can be used in all languages to obtain certain status information about the currently executing program. By its very nature this information will have been fixed at program start up, so no equivalent function to set these values is available, although they can all be controlled from the BLINKER environment variable.

Since BLIMGRSTS() can be called from any language, it uses the normal calling conventions for the language being used, and takes an integer parameter indicating the type of status information required. #defines or their equivalents are included in the header files BLX286.* for each of the major compilers and languages.

In the subsequent discussions and examples, the following terms are used:

Descriptor **A structure containing the size, base and attributes of a protected mode segment.**

Paragraph **The segment portion of a real mode address , e.g. the paragraph of the colour adapter is B800h.**

Segment **The actual code or data unit as allocated and stored in memory.**

Selector **The value used to identify a segment when running in protected mode, i.e. that is passed to and from the protected mode memory allocation functions.**

Calling conventions

All the API functions use the Pascal calling convention, except where otherwise noted. This means that the function arguments are pushed onto the stack from left to right, and the code calling the function is NOT responsible for removing them from the stack.

Far calls must be used with all the API functions, and all pointers must also be of type far.

All the API functions return an unsigned integer (short) in AX to indicate the success or failure of the function. A zero value indicates success and any other value is an error code. All other registers are preserved. If the function is required to return any other value or result, it will take as a parameter one or more pointers to variables in which to store the results.

Header files

Header files defining prototypes for the API functions and their parameters are provided for interfacing to C/C++, Pascal and Modula. These files can be found in the appropriate language subdirectory and are named BLX286.*.

The header files also contain definitions of any data structures required by the API, along with a number of macros to simplify their use.

BLINKER ENVIRONMENT OVERRIDE Specify that CLIPPER environment settings may override defaults

---

Purpose : **Specify that CLIPPER environment settings may override defaults**

Syntax   : **BLINKER ENVIRONMENT OVERRIDE**

As a default the variables set using the BLINKER EXECUTABLE CLIPPER command will not be overridden at application runtime. This allows multiple applications with different requirements to run on the same machine without changing the CLIPPER environment settings.

This command is used to tell Blinker to allow the CA-Clipper environment variables set using the BLINKER EXECUTABLE CLIPPER command to be overridden at application run time by the settings of the CLIPPER environment variable. When doing this you may want to consider renaming your CLIPPER environment variable with the command BLINKER ENVIRONMENT CLIPPER.

See Also:

Linking for protected mode

Overview

When converting a program to protected mode or dual mode the modifications to the link script file are minimal. Typically only two extra commands must be added to the link file, one to specify that the program is to be DOS extended and the other to include the Blinker library for the compiler being used.

Simply adding the command BLINKER EXECUTABLE EXTENDED (BLI EXE EXT) causes Blinker to create a DOS extended program. Similarly, adding the command BLINKER EXECUTABLE DUAL (BLI EXE DUA) causes Blinker to create a dual mode program.

When linking DOS extended or dual mode programs it is always necessary to include an additional Blinker library for the appropriate compiler. This library contains replacement .OBJs for any compiler library .OBJs which are not protected mode compatible.

The Blinker libraries all have names of the form BLXcccnn, where ccc indicates the vendor and compiler, and nn indicates the compiler version number. They are all compiled as large model and so should only be used with large model programs.

The appropriate compiler library must always be SEARCHed just before the standard compiler library or the script file containing the compiler library.

For example, with Microsoft Visual C++:

```
SEARCH BLXMVC10      # Blinker library
LIB LLIBCE           # MSVC library
```

or with CA-Clipper 5.2:

```
SEARCH BLXCLP52      # Blinker library
LIB CLIPPER          # CA-Clipper library
```

or:

```
SEARCH BLXCLP52      # Blinker library
@CL520MID            # Script containing CA-Clipper library
```

If the compiler library is not explicitly mentioned then place the SEARCH of the Blinker library as the last library in the link file.

When using third-party libraries please ensure that they are protected mode or dual mode compatible before attempting to use them in a DOS extended program. Blinkinc cannot guarantee that any particular library is either overlayable or protected mode compatible or both. Please refer to the file 3RDPARTY.DOC for up-to-date details on compatibility issues which we are aware of.

The third party libraries should typically be placed before the SEARCH of the Blinker library unless otherwise documented in the third party product.

Running and debugging protected mode programs

Overview

When Blinker creates a DOS extended program the format of the map file changes to reflect the different program structure. The segments are numbered sequentially starting at segment number 1, and many logical segments may be merged into a single physical segment.

Running in a multitasking environment is probably the easiest method of debugging, particularly as the operating systems provide some sort of task isolation and crash protection so it is not necessary to reboot each time a protection exception occurs.

When a protection exception or other protected mode error occurs, by default all the registers are dumped to the screen, along with the .EXE segment numbers associated with the segment registers. These segment numbers are the ones displayed in the MAP file created with the MAP S,A option, so can be used along with the offset registers IP, BX etc to quickly locate the location and cause of the error. For more details refer to Chapter 7 under .Debugging DOS extended programs..

The DOS extender has to perform much less work in a DPMI environment than it does otherwise, so you are strongly advised to thoroughly test your programs in all the different environments in which it is intended to run. These should certainly include DPMI, VCPI and XMS. If they are only available in the same environment as DPMI, then DPMI support can be disabled by setting the BLINKER environment variable as described below.

The actual mode in which the DOS extender is running can be tested using the DosGetHostMode() API function.

Link script commands to control runtime

Run time behaviour of a Blinker DOS extended program can be controlled at link time using various link script file commands. The current commands and their meanings are as follows (where there are choices the first setting shown is the default, and is in bold print):

```
BLINKER EXTMEM BLOCK          Set extended memory block size
BLINKER EXTMEM LIMIT nnnn     Set maximum extended memory to use
BLINKER HOST DPMI ON | OFF    Enable / Disable DPMI host support
BLINKER HOST VCPI ON | OFF    Enable / Disable VCPI host support
BLINKER HOST XMS ON | OFF     Enable / Disable XMS host support
```

See Also:

Transparent interrupt and function support

BIOS Interrupts

The following BIOS interrupt calls are handled transparently:

```
INT 10h  Video          Functions 00h -1Ch
INT 11h  Equipment list
INT 12h  Get mem size
INT 13h  Disk           Functions 00h -19h
INT 14h  RS232          Functions 00h -05h
INT 16h  Keyboard       Functions 00h -12h
INT 17h  Printer        Functions 00h -02h
INT 1Ah  Clock
INT 1Bh  Control Break
INT 1Ch  User Timer Tick
INT 33h  Mouse
INT 7Ah  IPX / SPX
```

DOS Interrupts

The following DOS interrupt calls are handled transparently :

```
INT 21h  Functions 00h - 6ch except :
   0FH   Open file (fcb)
   10H   Close file (fcb)
   14H   Sequential disk file read
   15H   Sequential disk file write
   16H   Create file
   21H   Random disk record read
   22H   Random disk record write
   23H   Get file size
   24H   Set random record field
   27H   Random block read
   28H   Random block write
   31H   TSR
   5DH   Miscellaneous DOS internal functions
   5EH   Microsoft Networks functions 3 and above
   5FH   Network functions 6 and above
   63H   Asian 2 byte character support
INT 23h  Control C
INT 25h  Absolute DOS Read
INT 26h  Absolute DOS Write
```

Module Definition Files

Overview

In addition to a free-format link script file to specify the .OBJ and .LIB files to be linked, Blinker uses a module definition file to provide extra information on the characteristics of a Windows .EXE. Since the Windows .EXE format is also used for DOS extended programs, many of the module definition file commands can also be used with DOS extended programs.

The format and commands in module definition files are identical to those used by Microsoft LINK, TLINK etc, with one or two additions to extend their functionality.

Blinker supports module definition files through the use of the

DEFFILE <filename>

link script command, with the default file type being .DEF.

The DEFFILE command can be placed anywhere in the link script file, but it is suggested that it be placed at the end to ensure that options specified therein take priority over those in the link script file.

Alternatively, to avoid maintaining two separate files, .DEF file commands may be included directly in the link script (.LNK) file between DEFBEGIN and DEFEND commands:

```
DEFBEGIN            // Begin .DEF commands
...
DEFEND              // End .DEF commands
```

Again these may be placed anywhere in the link script file, but it is suggested that they be placed at the end of the file.

Blinker looks for module definition files in the following order:

. In the path specified with the file name
. In the current directory
. In the LIB environment variable paths

For further information on module definition files beyond what is provided in this guide please see your compiler documentation.

Example of a .DEF file

A typical module definition file

```
NAME            MYPROG
DESCRIPTION     `Myprog Version 1.00'
EXETYPE         WINDOWS
STUB            `WINSTUB.EXE'
CODE            PRELOAD MOVEABLE DISCARDABLE
DATA            PRELOAD MOVEABLE MULTIPLE
HEAPSIZE        1024
STACKSIZE       8192
```

NAME sets a name for the program. It is usually the same name as the executable file.

DESCRIPTION allows a descriptive character string to be embedded in the program.

EXETYPE specifies which operating system the program will be running under.

STUB adds a DOS stub named `WINSTUB.EXE' to the beginning of the program.

CODE sets the default attributes of code segments. PRELOAD indicates that the segment is loaded when the program starts. MOVEABLE determines whether a segment can be moved in memory. DISCARDABLE determines whether a code segment can be discarded from memory to fill a different memory request.

DATA sets the default attributes of data segments. PRELOAD and MOVEABLE are as defined above.

MULTIPLE specifies that DGROUP is copied for each instance of the program.

To make a Windows .EXE four steps are required:

Compile the source code for Windows, as directed in your compiler manual.

Run the resource compiler to create any resources required, as directed in your compiler manual.

Run Blinker against the link script file and optional module definition file.

Run the resource compiler to add the resources to the end of the .EXE file.

.DEF file syntax

Keywords are not case sensitive.

A NAME or LIBRARY statement, if used, must be listed before all others.

If a statement is repeated with a different specification further on in the module definition file, the later statement overrides the earlier one.

Comments in a module definition file are denoted by a semicolon (;). The semicolon can appear anywhere on the line, and all information on the rest of that line is ignored.

If a string argument matches a reserved word it must be enclosed in double quotes (").

[^^Up^^]

[<<Previous Entry]

APPLOADER **Replace the default Windows loader with a custom loader**

Purpose : **Replace the default Windows loader with a custom loader**

Syntax : **APPLOADER 'loadername'**

Example : **APPLOADER 'MYLOADER'**

This optional command tells Blinker to use an externally defined loader as the identifying portion of the loader code segment name by prefixing it with LOADER_.

The command is intended to support self-loading FORTRAN programs, and must appear before any SEGMENTS commands.

CODE **Set default for CODE segment attributes**

Purpose : **Set default for CODE segment attributes**

Syntax : **CODE [attribute [, attribute ...]]**

Example : **CODE PRELOAD EXECUTEREAD**

The default attributes set with the CODE command can be overridden with the SEGMENTS command. Each attribute can be specified once in any order. This command should be listed before any instances of the SEGMENTS command in the definition file that refer to code segments.

For a complete list of applicable attributes please refer to the SEGMENTS command later in this chapter.

Purpose : **Set default DATA segment attributes**

Syntax : **DATA [attribute [, attribute ...]]**

Example : **DATA PRELOAD**

The default attributes set with the DATA command can be overridden with the SEGMENTS command. Each attribute can be specified once in any order. This command should come before any instances of the SEGMENTS command in the definition file that refer to data segments.

For a complete list of applicable attributes please refer to the SEGMENTS command later in this chapter.

Purpose : **Insert descriptive text into a program**

Syntax : **DESCRIPTION 'text'**

Example : **DESCRIPTION 'Professional Version 1.00'**

This command is used to embed a string of text up to 255 characters in length into a program .EXE or .DLL file. Note that by default the information is not used in any way by the created program.

This command is useful for inserting copyright or version information into a program.

Purpose : **Identify the target operating system**

Syntax : **EXETYPE [WINDOWS [n.nn] | OS2 | DOS]**

Example : **EXETYPE WINDOWS 3.1**

This command specifies the operating system under which the program is to be run.

When creating Windows programs, the version of Windows is important to ensure that the program will be able to run.

The default EXETYPE is WINDOWS, and the default Windows version number is 3.00.

Purpose : **Define exported functions**

Syntax : **EXPORTS entryname [=intname] [@ord [table]] [NODATA] [params]**

Example : **EXPORTS GetStr = String1**

This command makes listed functions (which are by default hidden) available to other executable files at run time.

entryname defines the function name as it is known to other programs.

intname defines the actual name of the function as it appears within the exporting program. This is by default the same as entryname.

ord defines a function's ordinal position within the module definition table.

table is one of two optional keywords RESIDENTNAME or NONAME which determine whether the entry name is placed in the resident or non-resident names table or discarded at link time. NONAME is the default and the most efficient way of exporting functions and means that the symbol can only be imported using the specified ordinal.

RESIDENTNAME allows the symbol to be directly imported by name.

NODATA specifies that there is no static data in the function, so the function will use the current data segment.

params specifies the number of words the function expects to be passed as parameters.

Purpose : **Specify function order and location**

Syntax : **FUNCTIONS [:segmentname] functionnames**

Example : **FUNCTIONS :seg1**

```
FUNCTIONS :seg1
func1
func2
func3
```

This command places functions contained in COMDAT records in a specified physical order, and can optionally assign them to a specified segment name.

BLINKER EXECUTABLE ALIGNMENT Specify the alignment size for a new .EXE file.

Purpose : **Specify the alignment size for a new .EXE file.**

Syntax   : **BLINKER EXECUTABLE ALIGNMENT <nuAlignment>**

Default : **16**

Example : **# Change alignment to 64 bytes for large .EXE**
BLINKER EXECUTABLE ALIGNMENT 64

This command specifies the alignment size, and therefore the maximum file size, for the new .EXE format files used for DOS extended and Windows programs. Segments and resources within a new .EXE file are referenced by 16 bit page numbers, which makes 65535 the maximum page number and 1 MB the default maximum .EXE size. Other linkers default to an alignment of 512 which can waste significant amounts of space inside the .EXE file.

The alignment can be any valid power of 2 above 16, but the larger the size the larger the wasted space in the .EXE file.

Alignment	Maximum .EXE Size
16	1 Mb
32	2 Mb
64	4 Mb
128	8 Mb

Purpose : **Set local heap size in bytes**

Syntax : **HEAPSIZE [size | MAXVAL]**

Example : **HEAPSIZE 3000**

This is used to set the size of the program's local heap in bytes. The heap is created in the default data segment after DGROUP, so its maximum size is determined by the size of DGROUP.

The keyword MAXVAL can be used to set the heap size to 64kb minus the size of DGROUP, i.e. the maximum possible size.

Purpose : **Define imported functions**

Syntax : **IMPORTS [intname=] modname.entry**

Example : **IMPORTS int1 = ext1.reader**

The IMPORTS command defines the names and location of functions to be imported for use in the current program. A definition is required for each function being imported.

intname is the function used in the importing program.

modname is the filename of the exporting program that contains the function.

entry can be either the name or the ordinal value of the function as defined in the modname program. If it is the ordinal value then the internal name must be specified.

Purpose : **Insert another file into the .DEF file**

Syntax : **INCLUDE filename**

Example : **INCLUDE OTHERFILE**

This command inserts the contents of a specified file into the .DEF file at the location of the INCLUDE command. The file to be inserted must contain valid module definition file statements. Included files may be nested up to 4 levels.

Purpose : **Create a DLL and name it**

Syntax : **LIBRARY [libname] [INITGLOBAL | INITINSTANCE]**
[PRIVATELIB] [WINDOWAPI]

Example : **LIBRARY mydll**

This command indicates that the resulting executable will in fact be a dynamic link library. If a name is not specified then the name of the executable file is taken.

The INITGLOBAL keyword tells Windows that it should keep one data area for all instances of the DLL, while the INITINSTANCE keyword tells Windows to keep separate data areas for each instance of the DLL. INITINSTANCE is the default.

The PRIVATELIB keyword tells Windows that only one program can use the DLL.

The WINDOWAPI keyword tells Windows that the DLL conforms to the Windows programming API, and is the default.

Purpose : **Create an application program and name it**

Syntax : **NAME [appname] [apptype] [NEWFILES]**

Example : **NAME myapp**

This statement identifies the executable file as a program as opposed to a DLL. If it is used then the LIBRARY command cannot also be used. If neither are specified then NAME is the default.

appname is the application name as it is known by the operating system, and cannot exceed 255 characters. If not specified, it defaults to the base name of the executable file.

apptype is one of three optional keywords:

WINDOWAPI. This is the default and creates a Windows-based program. The program uses the API provided by Windows and must be executed within Windows.

WINDOWCOMPAT. This option creates a character-mode program to run in a text window within a Windows session.

NOTWINDOWCOMPAT. This creates a character-mode program which must run full screen and cannot run in a text window within a Windows session.

The NEWFILES keyword notifies the loader that the program may be using an installable file system. The synonym LONGNAMES is supported for compatibility.

Purpose : **Preserve export information from a previous DLL**

Syntax : **OLD 'filename'**

Example : **OLD 'olderlib'**

When linking a DLL, this command instructs Blinker to search another DLL for export ordinals. Exported names in the current library which match those in the OLD library will receive the ordinal values from the OLD library.

For more information about ordinals for function identification please refer to the IMPORTS and EXPORTS commands.

Purpose : **Specify a protected mode Windows-based program**

Syntax : **PROTMODE**

This command specifies that the program should only run under Windows in protected mode. This command is assumed if EXETYPE has been set to WINDOWS.

Purpose : **Specify a real mode Windows-based program**

Syntax : **REALMODE**

This command specifies that the program can be run under Windows in real or protected mode. PROTMODE is assumed by default for EXETYPE WINDOWS.

Purpose : **Set attributes for specific segments**

Syntax : **SEGMENTS [']segname['] [CLASS 'classname']**
[attribute [attribute ...]]

Example : **SEGMENTS**
seg1 CLASS 'mycode' EXECUTEONLY
seg2 CLASS 'mydata'

The SEGMENTS command sets attributes for one or more segments on a segment by segment basis, overriding the defaults set by the CODE and DATA commands. The sequence of specified segments establishes their order in the output file. The segment class defaults to 'CODE' if CLASS 'classname' is not specified.

The following attributes may be specified , with the defaults shown first for each set of options :

discard [NONDISCARDABLE | DISCARDABLE]

This is used only for CODE and SEGMENTS commands. It determines whether a code segment can be discarded from memory to fill a different memory request.

execute [EXECUTEREAD | EXECUTEONLY]

This is used only for CODE and SEGMENTS commands. It determines whether a code segment can be read as well as executed. The EXECUTEREAD option is necessary for a program to run under a debugger which processes CodeView information.

instance [MULTIPLE | SINGLE | NONE]

This is used for the DATA command only. It affects the sharing attributes of DGROUP, the default data segment. This option interacts with the shared attribute.

MULTIPLE, the default for programs, specifies that DGROUP is copied for each instance of the program.

SINGLE, which is the default for DLLs, specifies that one DGROUP is shared by all instances of the DLL or program.

NONE tells the loader not to allocate DGROUP. This is used when a DLL has no data and uses the program's DGROUP.

This option interacts with the shared attribute for data segments.

load [LOADONCALL | PRELOAD]

This is used for CODE, DATA and SEGMENTS commands. It determines when a segment is loaded. LOADONCALL specifies that the segment should not be loaded until accessed, and only if not already loaded. PRELOAD specifies that the segment should be loaded when the program starts.

moveable [FIXED | MOVEABLE]

This is used for CODE, DATA and SEGMENTS commands. It determines whether a segment of a Windows program can be moved in memory. This command can also be used to force segments of a DOS extended program to be loaded in conventional memory.

read [READWRITE | READONLY]

This is used for DATA and SEGMENTS commands only. It determines access rights to a data segment.

shared [NONSHARED | SHARED]

This is used for real-mode Windows sessions only. It determines whether all instances of the program can share EXECUTEREAD and READWRITE segments.

NONSHARED, which is the default for programs, specifies that the segment must be loaded seperately for each process. An alternative keyword is IMPURE.

SHARED, which is the default for DLLs, specifies that only one copy of the segment should be loaded and shared among all processes that access the program or DLL. This setting saves memory and can be used for code that is not self-modifying. An alternative keyword is PURE. This option interacts with the instance attribute for data segments.

STACKSIZE **Set stack size in bytes**

Purpose : **Set stack size in bytes**

Syntax : **STACKSIZE n**

Example : **STACKSIZE 4096**

The number specified for the stack size must be a positive decimal integer.

[Next Entry>>]

STUB   Add a DOS stub to the beginning of the program file

Purpose : **Add a DOS stub to the beginning of the program file**

Syntax   : **STUB ['filename' | NONE]**

Example : **STUB 'MYSTUB.EXE'**

The stub added to the program will be called whenever the program is run under DOS. Normally it will display a warning message and terminate execution of the program.

A small stub program to do just this is automatically added to each Windows .EXE created by Blinker when no STUB command has been specified.

The PATH environment variable will be searched if the specified file is not in the current directory.

The keyword NONE causes the .EXE to be created without a stub program, thus making a slightly smaller executable. However, if the .EXE is run from the DOS prompt the system will most likely lock up, so this is not advised.

Purpose : **Specify default CLIPPER environment settings**

Syntax : **BLINKER EXECUTABLE CLIPPER <environment string>**

Example : **BLINKER EXECUTABLE CLIPPER //F:35**

This command is used to burn default values for the CA-Clipper runtime environment variables into the executable file.

Those settings set using the BLINKER EXECUTABLE CLIPPER command take precedence over those set with the SET CLIPPER command. Parameters not specified with the BLINKER EXECUTABLE CLIPPER command will retain values originally assigned by the SET CLIPPER command. However, if you want the values in the SET CLIPPER command to take precedence, use the command BLINKER ENVIRONMENT OVERRIDE.

The maximum string length that can be burned into a CA-Clipper application is 128 bytes, and Blinker will ignore all but the LAST environment burn-in command encountered, so all parameters that are to be burned in should be specified in a single BLINKER EXECUTABLE CLIPPER command.

PLEASE NOTE THAT THIS AND OTHER SIMILAR BLINKER COMMANDS TAKE EFFECT ONLY WHEN A FULL LINK IS EXECUTED.

The burned-in environment values are not returned by the CA-Clipper functions GETE() / GETENV().

Note for CA-Clipper 5.x:

The CA-Clipper runtime parameters listed here are applicable to CA-Clipper 5.x. Check the latest CA-Clipper documentation for a complete list of these parameters, correct syntax and default values.

E:nnnn	Expanded Memory (in kb)
F:nnn	Number of files
X:nn	eXclude memory in kb
BADCACHE	protect the EMS pageframe
CGACURS	disable EGA cursor
DYNF:nn	number of handles for dynamic paging
NOIDLE	disable background garbage collection
SWAPK:nnnn	size of disk swap file
SWAPPATH:'path'	path for swap file
TEMPPATH:'path'	path for temporary files

CA-Clipper 5.x environment settings can always be overridden (regardless of the BLINKER ENVIRONMENT OVERRIDE setting) on the application command line, using the command line switch '//' as in:

```
MYAPP //SWAPPATH:'D:\'
```

which would cause CA-Clipper to create the VMM swapfile in the root of drive D:.

There is usually no reason to disable expanded memory support (E0) for CA-Clipper 5.x applications.

Note for CA-Clipper Summer '87:

The CA-Clipper runtime settings are detailed in Chapter 9 - The Runtime Environment - of the CA-Clipper Summer '87 manual.

They consist of 6 values:

Ennn	Expanded memory (in kb)

```
Fnnn     Number of open files
Rnnn     Run command / index memory (in kb)
Sn       Snow indicator
Vnnn     Variable table area (in kb)
Xnnn     eXclude memory (in kb)
```

For example, to ensure that 41 files are available to CA-Clipper and that 100kb will be available to run external commands, use the following command:

BLINKER EXECUTABLE CLIPPER F41;R100

Remember that for the application to be able to use 41 file handles, corresponding values have to be set in the CONFIG.SYS, and, if necessary, in the appropriate network configuration.

At runtime, using the above burned-in settings, the 'F' and 'R' parameters in the CLIPPER environment will be ignored, but all the other parameters will be used in the normal way. It should be noted that the CLIPPER environment variable should be used to disable expanded memory usage by CA-Clipper Summer '87. If this is not done index files may be corrupted when CA-Clipper Summer '87 uses this memory in conjunction with certain expanded memory drivers, or RUN errors may occur. To disable CA-Clipper Summer '87 usage of expanded memory set the CLIPPER environment variable 'E' to 0 as follows (this may be used in conjunction with the other settings for number of files etc.):

SET CLIPPER=E0

Alternatively, the E0 may be burned into the application:

BLINKER EXECUTABLE CLIPPER E0

See Also:

Purpose : **Specify the creation of a dual mode executable.**

Syntax : **BLINKER EXECUTABLE DUAL [<minimum extended memory>]**

Default : **<minimum extended memory> is 512KB**

Example : **# Run in protected mode if at least 2MB available**
BLINKER EXECUTABLE DUAL 2048

This command specifies that Blinker should produce a dual mode executable that can be run in either real or protected mode. The optional parameter to this command is the minimum amount of available extended memory required to execute in protected mode.

At runtime, the program will be executed in REAL mode if any of the following are true:

- The CPU is less than an 80286
- The CPU is an 80286 or better, but there is no DPMI, VCPI or XMS host.
- The CPU is an 80286 or better, but the amount of available extended memory is less than the specified minimum amount.

Otherwise the program will be executed in protected mode.

If no BEGINAREA and ENDAREA commands are used with BLINKER EXECUTABLE DUAL then ALL overlaying will be disabled, which will create a non-overlaid program which will run in both real or protected mode.

Note: Dual mode programs must be specifically written to work in both real and protected mode. Use of this command does not imply that the program will run successfully as a dual mode program, it simply sets the format of the .EXE accordingly.

For a detailed description of the requirements of a dual mode program, see Chapter 5 'Coding for Dual mode'.

BLINKER EXECUTABLE EXTENDED   Specify the linking of a DOS extended executable

---

Purpose : **Specify the linking of a DOS extended executable**

Syntax : **BLINKER EXECUTABLE EXTENDED [<minimum extended memory>]**

Default : **Default for <minimum extended memory> is 512 Kb**

Example :
```
# Create a DOS extended program that
# requires 2 Mb extended memory
BLINKER EXECUTABLE EXTENDED 2048
```

This command specifies that Blinker should produce a protected mode DOS extended executable.

The executable generated will only run on an 80286 processor or above, with a minimum of 1 Mb of available extended memory.

If BLINKER EXECUTABLE EXTENDED is specified then ALL link script commands to do with overlaying are ignored, and a non-overlaid program is always created.

Note: Protected mode programs must be specifically written to work in protected mode. Use of this command does not imply that the program will run successfully as a protected mode program, it simply sets the format of the .EXE accordingly.

Purpose : **Specify number of IPX ecbs and send buffers**

Syntax : **BLINKER EXECUTABLE IPX <ecb count>[,<send buffer count>]**

Default : **64,32**

Example : **# Specify 48 ecbs and 16 send packet buffers**
BLINKER EXECUTABLE IPX 48,16

The Blinker Dos extender provides transparent Novell IPX and SPX support (Novell network communications), which requires low memory buffers to be allocated by the extender for both send and receive events.

This command specifies the number of IPX ECBs and send buffers that the DOS extended executable will be able to use.

Two types of buffers are required by the IPX/SPX module. The first type are the ECBs (event control blocks) one of which is needed for each pending send or receive event. The second type of buffers are the send packet buffers which hold data which has been submitted to IPX/SPX, but has yet to be sent (pending send events). In the case of ECBs, the number required is the total number of pending events that have been submitted to IPX, but have not yet completed at any given time. The number of send buffers required is the total number of simultaneous active send events.

Each ECB buffer specified requires approximately 42 bytes of conventional memory, and each send buffer specified requires approximately 550 bytes. The total size of both types of buffers may not exceed 64K when added together.

Unless your program posts a large number of IPX listens, or sends a large number of packets simultaneously, the default values should be sufficient.

No memory is allocated by the extender until the first IPX call is made.

Purpose : **Specify the linking of a real mode executable**

Syntax : **BLINKER EXECUTABLE REAL**

Example : **# Create a real mode only program**
BLINKER EXECUTABLE REAL

This command specifies that Blinker should produce a real mode executable, and is the default if no other executable type is specified.

Real mode executables may optionally contain dynamic overlays, but make no use of protected mode, so will run on all 80x86 processors.

BLINKER EXECUTABLE NOBLINK   Causes NO Blinker code whatsoever to be linked in

---

Purpose : **Causes NO Blinker code whatsoever to be linked in**

Syntax   : **BLINKER EXECUTABLE NOBLINK**

Default : **Disabled**

This command causes Blinker to ignore all overlays, demo restrictions, the CA-Clipper 5.x paging system, and will give unresolved externals for all Blinker functions including the swap functions. This is mainly used in order to create small C / ASM programs with no overhead.

DO NOT USE FOR CA-CLIPPER EXCEPT FOR SMALL PROGRAMS WITHOUT OVERLAYS !!

BLINKER EXECUTABLE NODELETE   Create .EXE regardless of unresolved externals

---

Purpose : **Create .EXE regardless of unresolved externals**

Syntax   : **BLINKER EXECUTABLE NODELETE**

Default : **Disabled**

This command is used to prevent Blinker from deleting the .EXE file even when there are unresolved externals. This can be useful during the development cycle, when not all routines have been defined. Note that at runtime the program will fail if an unresolved external is called.

BLINKER EXECUTABLE SERIAL   Specify serial number to encrypt into the .EXE

Purpose : **Specify serial number to encrypt into the .EXE**

Syntax   : **BLINKER EXECUTABLE SERIAL <cSerialNumber>**

Example : **BLINKER EXECUTABLE SERIAL 001001 5 Blink, Inc.**

This command is used to encrypt a serial number and other information of up to 100 characters into the executable file which can then be examined by the program using the BLISERNUM() function at runtime.

This information is encrypted into the executable file and can be used to hide a serial number and licensee name in the executable file as a form of copy protection. Any information following the BLINKER EXECUTABLE SERIAL command on the same line of the script file is encrypted into the .EXE file up to a limit of 100 characters. An identical string is returned in the same format by the BLISERNUM() function at run time.

The serial number in the above example can be examined by assigning the serial number to a string and then manipulating it using a substring function, for example:

```
TMPSTRVAR = BLISERNUM()
? "Serial number :", SUBSTR(TMPSTRVAR,1,8)
? "Number of users :", VAL(SUBSTR(TMPSTRVAR,10,1))
? "Licensed to :", SUBSTR(TMPSTRVAR,12,20)
```

See Also:

BLINKER EXTMEM BLOCK   Specify the extended memory allocation block size

----

Purpose : **Specify the extended memory allocation block size**

Syntax   : **BLINKER EXTMEM BLOCK <nuSize>**

Default : **256**

Example : **# Allocate in 512KB blocks**
BLINKER EXTMEM BLOCK 512

This command specifies the memory block size (in KB) used by the extender when allocating extended memory blocks. When running in protected mode, the extender allocates memory from the host in blocks of the specified size and manages these blocks itself.

When a DOS extended program requests memory, it is allocated memory from these extender managed blocks, rather than directly from the host. Programs which allocate large amounts of memory may benefit from a larger block size than the 256 KB default.

Note also that the BLINKER environment variable setting /EBnnnn will override this function.

See Also:

Purpose : **Specify the total extended memory to be allocated by the extender**

Syntax : **BLINKER EXTMEM LIMIT <nuSize>**

Default : **All available extended memory**

This command specifies the total amount of physical memory the extender will allocate before the extender's VM system begins to swap to disk in order to satisfy new allocation requests. This command can be used to ensure that a DOS extended program does not use up all the available memory unnecessarily, or to simulate running in lower memory conditions.

Note also that the BLINKER environment variable setting /ELnnnn will override this function.

See Also:

BLINKER HOST DPMI **Enable / disable the extender's use of a DPMI host.**

Purpose : **Enable / disable the extender's use of a DPMI host.**

Syntax : **BLINKER HOST DPMI ON | OFF**

Default : **ON (enable use of DPMI)**

This command can be used to disable the extender's use of a DPMI host for testing under non-DPMI hosts, or to disable use of a DPMI host if problems are suspected.

Under normal circumstances, the extender checks for hosts in the following order:

DPMI
VCPI
XMS

Note also that the BLINKER environment variable setting /HDc will override this setting.

See Also:

BLINKER HOST MESSAGE   Enable / disable the extender's host display.

Purpose : **Enable / disable the extender's host display.**

Syntax   : **BLINKER HOST MESSAGE ON | OFF**

Default : **OFF (disable host messages)**

This command can be used to enable the display of various informational messages concerning the available hosts at program startup.

The command provides a convenient method of checking the current host, processor type, amount of memory available, and Blinker version number when an extended or dual mode program starts execution:

For example:

```
BLX286 : EMS/XMS sharing present
BLX286 : DPMI host not detected
BLX286 : XMS host detected
BLX286 : VCPI host detected
BLX286 : DPMI host emulator initialised
BLX286 : Active host is VCPI (v3.01 i486 5668 kb)
```

Note also that the BLINKER environment variable setting /HMc will override this setting.

BLINKER HOST PASSUP   Enable / disable automatic pass up of IRQ interrupts.

Purpose : **Enable / disable automatic pass up of IRQ interrupts.**

Syntax   : **BLINKER HOST PASSUP ON | OFF**

Default : **ON (enable passup of IRQ interrupts)**

This command can be used to enable or disable the extender's automatic pass up of IRQ interrupts from real mode to protected mode.

This command should only be necessary for debugging purposes when testing interrupt handlers.

BLINKER HOST QDPMI   **Enable / disable the extender's use of a QDPMI host.**

Purpose : **Enable / disable the extender's use of a QDPMI host.**

Syntax   : **BLINKER HOST QDPMI ON | OFF**

Default : **ON (enable use of QDPMI)**

This command can be used to disable the extender's use of a QDPMI host for testing under non-QDPMI hosts, or to disable use of a QDPMI host if problems are suspected.

Under normal circumstances, the extender checks for hosts in the following order:

DPMI
VCPI
XMS

Note also that the BLINKER environment variable setting /HDc will override this setting.

See Also:

BLINKER HOST VCPI **Enable / disable the extender's use of a VCPI host.**

Purpose : **Enable / disable the extender's use of a VCPI host.**

Syntax : **BLINKER HOST VCPI ON | OFF**

Default : **ON (enable use of VCPI)**

This command can be used to disable the extender's use of a VCPI host for testing under a non VCPI host, or to disable use of a VCPI host if problems are suspected.

Under normal circumstances, the extender checks for hosts in the following order:

```
DPMI
VCPI
XMS
```

Note also that the BLINKER environment variable setting /HVc will override this setting.

See Also:

BLINKER HOST XMS **Enable / disable the extender's use of an XMS host.**

Purpose : **Enable / disable the extender's use of an XMS host.**

Syntax : **BLINKER HOST XMS ON | OFF**

Default : **ON (enable use of XMS)**

This command can be used to disable the extender's use of an XMS host for testing under a non XMS host, or to disable use of an XMS host if problems are suspected.

Under normal circumstances, the extender checks for and makes use of hosts in the following order:

```
DPMI
VCPI
XMS
```

Note also that the BLINKER environment variable setting /HXc will override this setting.

See Also:

BLINKER INCREMENTAL FILE   **Specify alternate file name for CA-Clipper incremental file**

Purpose : **Specify alternate file name for CA-Clipper incremental file**

Syntax : **BLINKER INCREMENTAL FILE [d:][path]<filename>**

Default : **<ExeName>.BIF**

Example : **# Create the .BIF file on RAM drive E**
BLINKER INCREMENTAL FILE E:Temp.BIF

This command is used to specify a different name for the incremental link file for a program. The default name used is the same as the .EXE file to be output, and the default extension used is .BIF.

Please ensure that the file specified does not already exist as an incremental file for a different program.

See Also:

BLINKER INCREMENTAL OFF   Disable CA-Clipper incremental linking

----

Purpose : **Disable CA-Clipper incremental linking**

Syntax : **BLINKER INCREMENTAL OFF**

Default : **ON**

This command is used to turn off Blinker's use of incremental linking. Every link performed will be a full link, no new incremental .BIF file will be created, and no padding will be left between procedures to allow room for incremental linking.

This command should be used for the final link of an application to obtain its smallest memory requirement, and best runtime performance.

Symbol Table Compression:

Using this command also enables symbol table compression for both CA-Clipper Summer'87 and CA-Clipper 5.x, which reduces the size of the application symbol table by removing unneeded duplicate entries.

The amount of reduction achieved may vary between individual applications, depending on programming style, and the size of the application.

Dynamic Paging:

Using this option with CA-Clipper 5.x applications also enables dynamic paging of CA-Clipper code, which provides very good runtime performance for overlaid CA-Clipper code, with negligible memory overhead.

See Also:

BLINKER INCREMENTAL PAD **Specify size of padding for CA-Clipper incremental linking**

Purpose : **Specify size of padding for CA-Clipper incremental linking**

Syntax : **BLINKER INCREMENTAL PAD <nisize>**

Default : **128 bytes**

Example : **# Leave 200 bytes at end of every procedure**
BLINKER INCREMENTAL PAD 200

This command is used to set the size of the padding attached to the end of each procedure when it is first linked incrementally. The parameter is a decimal number indicating the size in bytes to be added at the end of each procedure in order to leave room for expansion, and may be any size up to 1024. (The default value is 128, which should leave a reasonable amount of padding for the majority of simple changes made during final development of a routine.)

Increasing this value allows the application to link incrementally more often without having to do a full link, but it will reduce the amount of memory available to the application at runtime during testing of real mode programs.

See Also:

BLINKER LINK EMS **Specify link time EMS memory use**

Purpose : **Specify link time EMS memory use**

Syntax : **BLINKER LINK EMS ON | OFF**

Default : **ON**

Example : **# Disable EMS due to a conflict with a TSR**
BLINKER LINK EMS OFF

This command is used to control whether EMS memory is used at link time by the virtual memory system within Blinker. There is usually no reason to use this command unless a conflict with some other software is suspected.

This command also sets BLINKER LINK PAGEFRAME OFF.

See Also:

```
 BLINKER LINK PAGEFRAME        Specify link time pageframe use
------------------------------------------------------------------------------

 Purpose  : Specify link time pageframe use

 Syntax   : BLINKER LINK PAGEFRAME ON | OFF

 Default  : ON

 Example  : # Disable pageframe due to a conflict with a disk cache
            BLINKER LINK PAGEFRAME OFF

 This command is used to control the use of the EMS pageframe by Blinker at
 link time by the virtual memory system within Blinker.

 There is usually no reason to use this command unless a conflict with some
 other software is suspected.
```

See Also:

Purpose : **Specify minimum Blinker version required**

Syntax   : **BLINKER LINK VERSION <n.n>**

Default : **0.0**

Example : **# Blinker 2.0 required to process this script**
           BLINKER LINK VERSION 2.0

This command is used to ensure a minimum version of Blinker to process the link script. If the version of Blinker is less than the specified minimum version number, Blinker will terminate with an error. Versions of Blinker prior to Blinker 2.0 do not recognize this command!

BLINKER LINK XMS **Specify link time XMS memory use**

Purpose : **Specify link time XMS memory use**

Syntax : **BLINKER LINK XMS ON | OFF**

Default : **ON**

Example : **#disable XMS use**
BLINKER LINK XMS OFF

This command is used to control whether XMS memory is used at link time by the virtual memory system within Blinker. There is usually no reason to use this command unless a conflict with some other software is suspected.

See Also:

Purpose : **Reserve conventional memory for program use**

Syntax : **BLINKER LOWMEM RESERVE <nuMemSize>**

Default : **<nuMemSize> defaults to 16 Kb**

Example : **# Reserve 32 Kb of conventional memory**
BLINKER LOWMEM RESERVE 32

This command is used to reserve the specified amount of conventional memory for program use, preventing the extender from using it for other purposes.

Under normal circumstances the extender will use all available extended memory, followed by all available conventional memory, to satisfy program memory requests. This may result in there beeing no conventional memory left for critical routines that require conventional memory to communicate with TSRs, real mode interrupt handlers or other real mode code.

Memory reserved by this command can be allocated by the program using DOSAllocRealSeg() or DPMI function 0x100.

Some third party libraries may require a BLINKER LOWMEM RESERVE setting for correct operation - consult the library documentation or contact the library vendor for further information.

BLINKER MEMORY CLEAR   Clears all memory beyond the EXE file to a specified value

----

Purpose : **Clears all memory beyond the EXE file to a specified value**

Syntax   : **BLINKER MEMORY CLEAR <nuValue>**

Default : **Disabled**

Example : **# Clear memory to NULLs**
BLINKER MEMORY CLEAR 0

This command causes Blinker to clear all of memory above the stack through to the top of memory with the decimal value specified (0 - 255) for DOS real mode programs. This can be used to help identify problems with uninitialised memory etc., and programs should not normally be shipped with this option enabled. Memory is cleared before the program starts executing.

Purpose : **Specify CA-Clipper Summer '87 memory packing frequency**

Syntax   : **BLINKER MEMORY PACK <nuFrequency>**

Default : **0**

Example : **# Set the packing to once every 20 overlaid procedure calls**
BLINKER MEMORY PACK 20

This command is used to set the frequency with which the Blinker overlay manager defragments CA-Clipper Summer '87 free memory. The parameter is a decimal number indicating the number of overlaid procedure calls after which memory is defragmented, and may be set to any value from 0 to 65534. (The default value is 0, which causes no packing to occur.)

CA-Clipper is continually allocating and freeing dynamic memory, causing it to become more and more fragmented as execution of an application continues. CA-Clipper only considers memory to be available if it is in chunks of 4kb or greater, thus large applications can often run out of memory after prolonged execution.

The BLINKER MEMORY PACK command allows the developer of a CA-Clipper Summer '87 application to defragment dynamic memory as often as needed without modifying the application, and to tune the frequency of the packing to suit the individual application. This process is also performed by the CA-Clipper function MEMORY(0), but much more slowly.

The defragmentation process consists of two things: joining up of contiguous blocks of free memory, and then resetting the roving pointer which is used in selecting (indicating) the next memory allocation. At the time of the defragmentation, this pointer is set to specify the highest available location in memory. This causes memory to be allocated from the beginning each time rather than fragmenting the whole memory area, which can significantly reduce the deterioration of available memory due to fragmentation.

Due to the underlying architecture of CA-Clipper Summer '87's memory allocation scheme, it is not possible to move allocated blocks of memory around, as there is no means of updating the pointer(s) to allocated blocks. This means that true garbage collection is impossible, but the defragmentation described here is the next best thing, and helps the problem considerably.

Setting the frequency to a low number may slow down the application, but will keep memory fragmentation to a minimum, giving the maximum saving on memory. Setting the frequency to a higher number will not noticeably slow the application down, but the memory gain will be less as the fragmentation will be relatively worse (although still a lot better than with no packing at all.)

An optimum frequency appears to be around 10 for an average application, giving a significant increase in memory while not significantly slowing the application. The optimum frequency for a given application may vary considerably from this frequency, and can only be established by trial and error.

See Also:

BLINKER MESSAGE DUPLICATES   Enable warnings for duplicate modules within libraries

----

Purpose : **Enable warnings for duplicate modules within libraries**

Syntax   : **BLINKER MESSAGE DUPLICATES**

Default : **Disabled**

This command causes Blinker to generate the 1004 link time warning message for all duplicate symbols encountered at link time. This command can be a very useful debugging tool for identifying duplicate modules among the files and libraries used in linking a program. The first module listed in the Blinker message 1004 is the module which is actually linked. The second module in the 1004 message is the duplicate and is ignored unless overridden with the MODULE FROM command.

The following modules (and included symbols) are processed at link time and listed in the first half of the VERBOSE output. Each of these symbols will be listed in the generated 1004 messages if another processed symbol has the same name.

. All modules specified in a FILE command.
. Those modules extracted from a library to resolve external references.
. All modules within a SEARCHed library.

Note that there may be numerous duplicated modules or symbols in the included libraries which are not reported when BLINKER MESSAGE DUPLICATES is used, since one or both of the modules are not processed.

For optimal use of this command, it will help to understand the section "The Linking Process" in Chapter 3, which is a discussion of the method by which Blinker processes the files and libraries included in a single link. This method determines which modules are processed, and hence which modules are identified as duplicates in the output generated by the BLINKER MESSAGE DUPLICATES command.

To obtain a list of all duplicated modules relevant to your program by using this command in conjunction with the VERBOSE command:

(1) List all possible duplicates:

The first half of the VERBOSE output will identify all modules processed during linking. The duplicates among these modules will be reported by the BLINKER MESSAGE DUPLICATES command. Since you can force all modules within a library to be processed by SEARCHing that library, SEARCHing all included libraries will cause all modules within the program to be processed, and hence cause all duplicates to be listed. Linking with all these SEARCHes is only intended to produce the desired duplicate messages, not a viable .EXE.

(2) Identify only those modules used in the program:

The second half of the VERBOSE output will identify only those modules actually included in the final executable file. The symbols highlighted as duplicates whose modules appear in the second half of the VERBOSE listing are the ones to examine. Generally all duplicates between third party libraries or files which are provided by the same vendor are intended and can be ignored; also those duplicates between a third party product and a compiler library can be assumed to be intended.

See Also:

BLINKER MESSAGE NOBLINK **Disable display of blinking eyes**

Purpose : **Disable display of blinking eyes**

Syntax : **BLINKER MESSAGE NOBLINK**

Default : **Disabled**

This command is used to switch off the blinking eyes which are displayed while Blinker is running.

See Also:

BLINKER MESSAGE NOWARNING   Disable display of link time warning messages

---

Purpose : **Disable display of link time warning messages**

Syntax   : **BLINKER MESSAGE NOWARNING**

Default : **Disabled**

This command is used to suppress the display of Blinker link time warning messages 1001 to 1100.

BLINKER MESSAGE WINK   Enable single eye winking during linking

----

Purpose : **Enable single eye winking during linking**

Syntax   : **BLINKER MESSAGE WINK [LEFT]**

Default : **Right winking eye**

Example : **# Display a winking right eye**
BLINKER MESSAGE WINK

This command is used to enable the winking of only one eye while Blinker is running. If no parameter is entered, the right eye will wink (from Blinker's point of view, i.e. from inside the screen looking out). The parameter LEFT causes the left eye to wink.

See Also:

Purpose : **Specify CA-Clipper Summer '87 dynamic overlay allocation method**

Syntax : **BLINKER OVERLAY DYNAMIC**

Default : **Enabled for CA-Clipper Summer '87**

This command is used to instruct Blinker to dynamically allocate memory for overlays from the CA-Clipper free pool on an as-needed basis. This is the default method of memory allocation for CA-Clipper Summer '87 overlaid applications, and was the only method used by Blinker versions up to and including Blinker 1.31.

This method has the benefit that the memory allocated to inactive overlays can be returned to CA-Clipper using the BLIOVLCLR() function to clear out inactive overlays. The disadvantage of this method is that it cannot make use of UMBs or the EMS pageframe for running overlays, so can leave less memory for the application under certain circumstances.

See Also:

BLINKER OVERLAY FIXED   Specify fixed overlay allocation method

----

Purpose : **Specify fixed overlay allocation method**

Syntax   : **BLINKER OVERLAY FIXED**

Default : **Enabled (Optional for CA-Clipper Summer '87)**

This command is used to instruct Blinker to allocate a fixed size block of memory from DOS for overlays at application start up. Once the block has been allocated from DOS, the Blinker overlay manager will not allocate any further memory for overlays, resulting in an application with a predictable overlay overhead. This method allows parental overlays (procedures or functions that called the current procedure) to be swapped out as required, in order to make room to execute new procedures.

This method is the only overlay pool allocation method for all compilers except CA-Clipper Summer '87, where it is optional.

Please see the previous section, BLINKER OVERLAY DYNAMIC, for a comparison of the FIXED and DYNAMIC methods.

See Also:

Purpose : **Request a size for the overlay pool**

Syntax : **BLINKER OVERLAY OPSIZE <niSize>**

Default : **40 kb**

Example : **# Increase the overlay pool size to 60kb**
BLINKER OVERLAY OPSIZE 60

This command is used to set the operating size of the overlay pool at run time in kb. It can be set to any value between 12 and 128. The default operating size is 40kb. This may be made smaller to save memory or larger to reduce the number of disk accesses made by the overlay manager to re-load overlaid procedures.

When overlays are specified in the link script file, Blinker will determine the size of the smallest possible overlay pool by adding approximately 8kb to the size of the largest overlay unit (a segment for most compilers). This minimum value will be displayed along with the value specified in the BLINKER OVERLAY OPSIZE command. If the minimum value is greater than the value specified, the minimum value will override the value specified.

The overlay opsize may be adjusted at runtime through the use of the BLINKER environment variable (see the 'Environment Variable' section in Chapter 2). For a discussion of how the overlay pool is used at runtime, see the 'Overlay Pool' section in Chapter 4.

Notes for CA-Clipper Summer '87:

When using CA-Clipper Summer '87 and the DYNAMIC overlay allocation method (see BLINKER OVERLAY DYNAMIC), the minimum OPSIZE is not displayed as it cannot be calculated at link time. The actual OPSIZE required while executing the program can be determined by setting it to a small value such as 10kb, running through all the options of the application WITHOUT performing a BLIOVLCLR(), and then checking the value of the Blinker function BLIOVLOPS() just before quitting the program. The value returned by the BLIOVLOPS() function will be the minimum value for the OPSIZE.

The reason for this is that with the DYNAMIC method all active overlays must remain in memory at a given time. The minimum overlay OPSIZE is therefore the total size of the active overlays when the program is at its deepest level of nesting. The method described above may be used to establish this value, and setting it to a smaller value will simply cause the value to be overridden at runtime.

See Also: